**UD 02 - PROGRAMACIÓN MULTIHILO**

1. **Conceptos de programación multihilo y de hilo**

Icono

Descripción generada automáticamente con confianza mediaSegún el número de flujos de ejecución que realice un programa los podemos clasificar en:

* Programa de flujo único (Secuencial): Realiza tareas una a continuación de otra, de forma secuencial. Es necesario esperar a que una acabe para seguir con otra.
* Programa de flujo múltiple (Multihilo): Coloca las actividades a realizar en diferentes flujos de ejecución, de forma que cada uno se inicia y termina por separado.

En la **PROGRAMACIÓN MULTIHILO (MULTITHREADING)** a cada flujo de ejecución se le llama thread o hilo.

Ejemplo: Tenemos un proceso (navegador web) en el que se hacen varias cosas al mismo tiempo como descargar una foto, descargar un pdf, seguir navegando por internet (cada una de ellas está en un hilo).

OJO: “A la vez” significa que se realizan concurrentemente (con la sensación de que el programa realiza de forma simultánea diferentes tareas porque el planificador les va dando turno a cada una para ejecutarse en la CPU de forma muy rápida). Para que de verdad fuesen en paralelo esto dependería del sistema operativo y del número de procesadores del sistema donde se ejecuta la aplicación.

**Definición de hilo o subproceso**: Flujo de control secuencial independiente dentro de un proceso. Está asociado con una secuencia de instrucciones, un conjunto de registros y una pila. Es controlado por el planificador que gestiona el tiempo de ejecución del procesador y asigna de alguna manera dicho tiempo a los diferentes hilos actualmente presentes.

Cuando se ejecuta un programa, el sistema operativo crea un proceso y su primer hilo (hilo primario) que a su vez puede crear hilos adicionales. Así:

* Un hilo no puede existir independientemente de un proceso, ni puede ejecutarse solo.
* Dentro de cada proceso puede haber varios hilos ejecutándose.
* Diagrama, Texto

  Descripción generada automáticamentePodríamos decir que no es el proceso el que se ejecuta, sino que es el espacio de direcciones donde reside el código el que es ejecutado mediante uno o más hilos.

El hilo es como un programa secuencial; no ofrece nada nuevo. Es la habilidad de ejecutar varios hilos en el proceso lo que da algo nuevo ya que permite la ejecución de diferentes actividades al mismo tiempo.

|  |  |
| --- | --- |
| El hilo posee como únicos: | Y puede compartir con otros hilos del mismo proceso: |
| * Identificador único * Contador de programa propio * Conjunto e registros * Pila (variables locales) | * Código * Datos (variables globales) * Recursos del sistema operativo (archivos abiertos, señales) |

**Ventajas y uso de hilos**

A los hilos se les ha llegado a llamar “procesos ligeros” porque frente a estos:

* Consumen menos recursos en el lanzamiento
* Se tarda menos tiempo en crearlos y terminarlos
* La conmutación entre hilos (cambio de contexto) es más rápida.

Se recomienda el uso de hilos cuando:

* La aplicación maneja entrada a varios dispositivos de comunicación
* Debe poder realizar diferentes tareas al mismo tiempo
* Interesa dar diferente prioridad a las tareas
* La aplicación se ejecuta en un entorno multiprocesador

Son ideales para entornos interactivos y en red, simuladores, animaciones. Todos los programas con interfaz gráfico son multihilo, porque los eventos y rutinas de dibujados de las ventanas corren en un hulo distinto al principal (AWT, Swing… usan hilos)

1. **Utilidades de Java para manejar los hilos**

Paquete java.lang

* Clase **Thread**: Responsable de producir hilos y de aportar métodos para su gestión.
* Interfaz **Runnable**: Capacidad de añadir la funcionalidad de hilo a una clase.
* Clase **ThreadDeath**: Clase de error, derivada de Error queda medios para manejar y notificar errores.
* Clase **ThreadGroup**: Se usa para manejar un grupo de hilos de modo conjunto.
* Clase **Object**: De ella heredan todos los objetos de java. Da algunos métodos importantes en la arquitectura multihilo (**wait**, **notify** y **notifyAll**)
* Clase **Timer**: Permite la ejecución de tareas programables en diferido y de forma repetitiva mediante el método shedule. (Objetos de la clase TimerTask).

Paquete java.util.concurrent

* Clases de sincronización **Semaphore**, **CountDownLatch**, **CyclicBarrier** y **Exchanger**
* Interfaces para separar la lógica de la ejecución: **Executor**, **ExecutorService**, **ScheduledExecutorService, Callable** y **Future**
* Interfaces para gestionar las colas de hilos: **BlockingQueque**, **LinkedBlokingQueque**, **ArrayBlockingQueque**, **SynchronousQueque**, **PriorityBlockingQueque**, **DelayQueque**.

Paquete java.util.concurrent.atomic

* Clases para ser usadas como variables atómicas en clases multihilo: **AtomicInteger**, **AtomicLong**

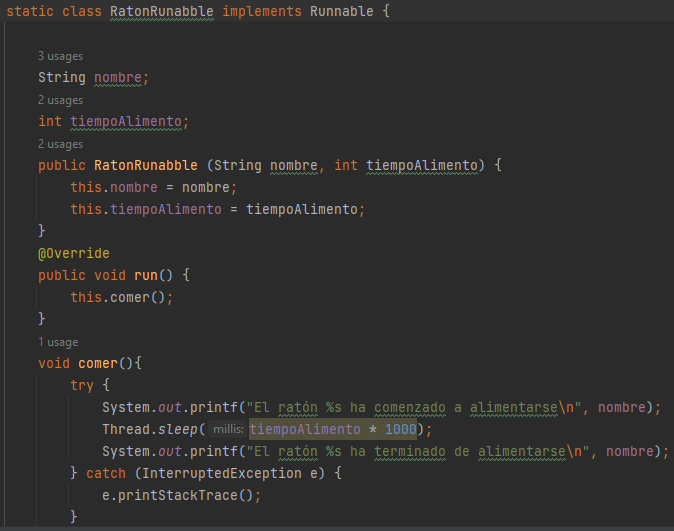
Paquete java.util.concurrent.locks

* Clases alternativas a la claúsula sinchronized. Interfaces como **Lock**, **ReadWriteLock**.

**3. Creación de hilos**

Para conseguir hilos de un objeto es necesario sobreescribir o implementar su método **run()**. El código en su interior es el que ejecutará el hilo. Para ello el objeto debe:

* Texto

  Descripción generada automáticamenteHeredar de **Thread**
* Implementar **Runnable**

Y ya podríamos instanciar los objetos que queremos, de un tipo o de otro:

Texto

Descripción generada automáticamente

Los hilos se crean llamando al método **start()** (**y no al método run directamente**, en cuyo caso la ejecución se seguirá produciendo de forma secuencial y no habrá ninguna ventaja).

Texto

Descripción generada automáticamenteSi lo hicimos heredando de Thread, solo vamos a poder ejecutar un hilo por objeto. Si intentamos llamar dos veces al método start() para el mismo objeto, saltará *IllegalThreadStateException*. (Llamar dos veces al start de mario, por ejemplo).

Imagen que contiene placa, texto, pequeño, tabla

Descripción generada automáticamenteSi lo hicimos implementando Runnable, podemos ejecutar todos los hilos que queramos para un mismo objeto. Aunque, para poder llamar al método, tendremos que encapsular el objeto dentro de un Thread. (También se puede hacer cuando se hereda de Thread esto de encapsular en un Thread para que no salte la excepción antes mencionada)

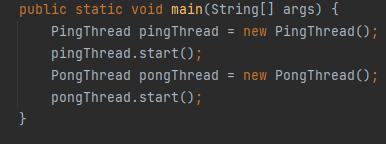
Texto

Descripción generada automáticamente

Cuando hablamos de procesos, el sistema operativo protege a los demás procesos de la corrupción de memoria provocada por uno de ellos. Esto no sucede así en los hilos. Pueden surgir errores de concurrencia si estos hilos trabajan sobre únicas copias del objeto en memoria y tienen variables compartidas. (Por ejemplo, si contamos el alimento consumido y lanzamos 100 hilos es muy posible que no dé 100 sino un número inferior). De hecho, es necesario desarrollar esquemas de bloqueo y sincronización para programar bien con hilos, siendo decisivos para obtener una aplicación eficiente o ineficiente. Escala de tiempo

Descripción generada automáticamente

El método estático **Thread.yield()** le dice al planificador que cambie de hilo. Con esto podríamos simular (muy rudimentariamente) un ping-pong.



Cuando la JVM arranca la ejecución de un programa, ya hay un hilo ejecutándose (hilo principal del programa) controlado por el método main. Este hilo que se ejecuta cuando el programa comienza es el último hilo en terminar su ejecución.

Para saber qué hilo se ejecuta en cada momento se puede usar el método **currentThread** y obtener su nombre mediante **getName**



Texto

Descripción generada automáticamente

(Vemos que el resto de instrucciones del main siguen ejecutándose una vez iniciados los hilos. Esto es porque no se ha indicado el método join() que veremos más adelante)

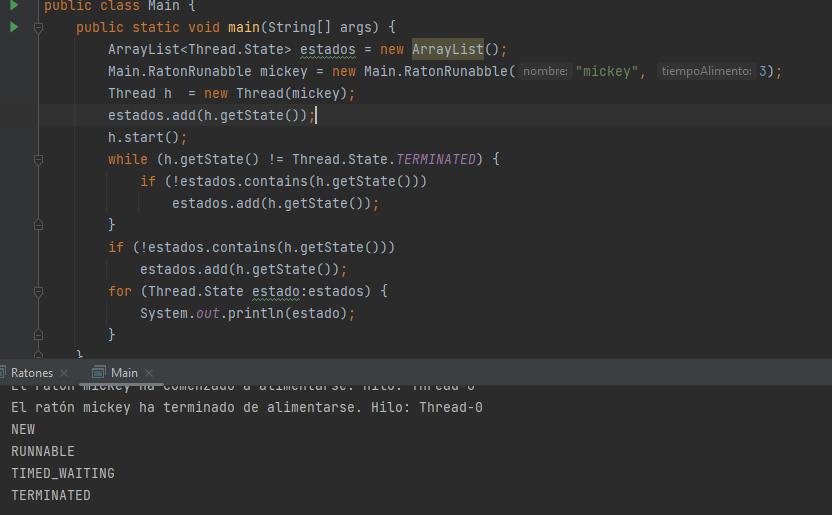
1. **Estados de un hilo: Inicio, detención, finalización**

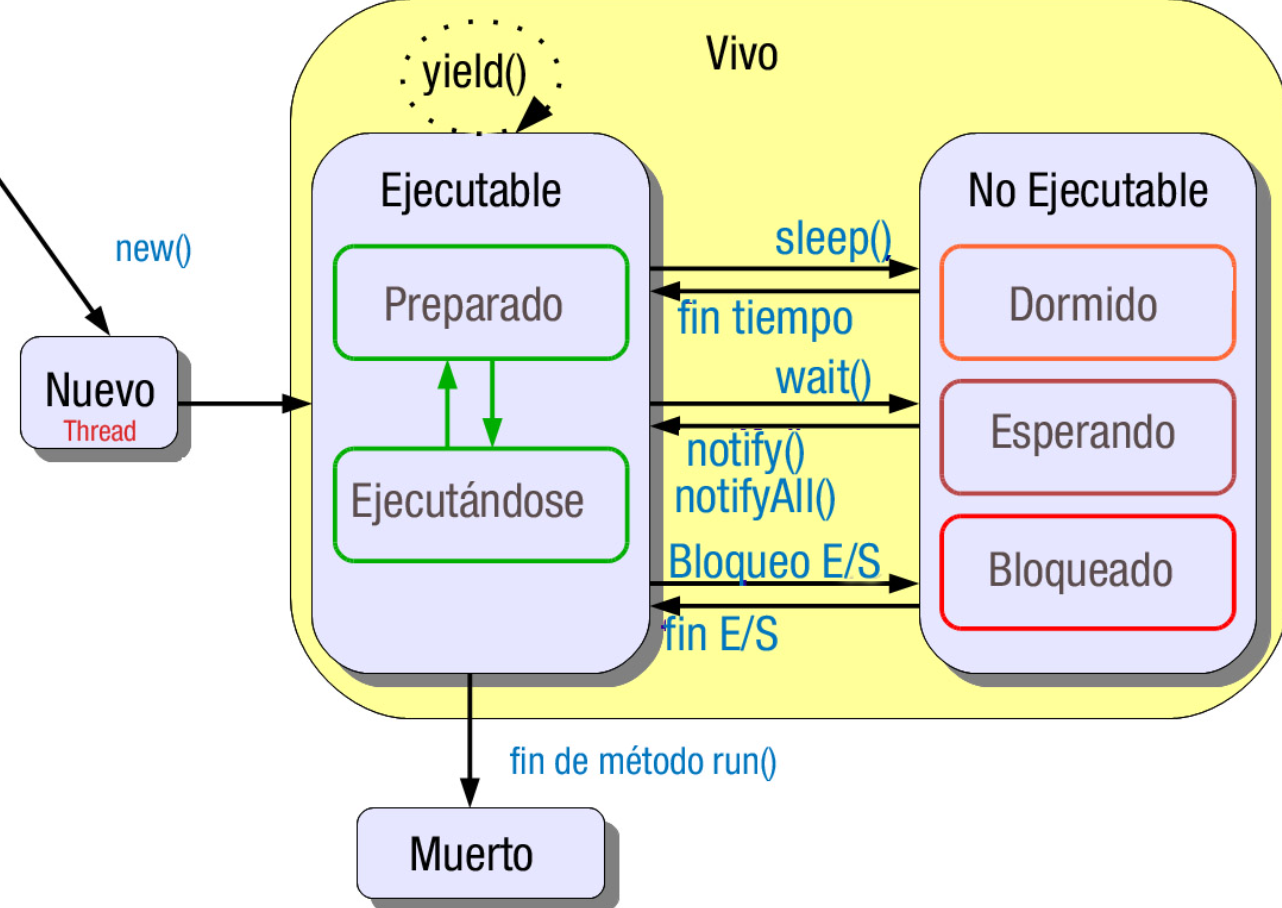
En java los estado un hilo se recogen en el enumerado State (clase java.lang.Thread);

* **NEW: Nuevo**. Hilo creado pero no disponible para su ejecución (no arrancado)
* **RUNNABLE: Ejecutable**. Hilo preparado para ejecutarse. Puede estar ejecutándose si se le ha asignado tiempo de procesamiento o no ejecutándose en beneficio de otro hilo, en cuyo caso está **preparado**.
* **BLOCKED**: **Bloqueado**. Bloqueado por un monitor.
* **WAITING**: **Esperando**. Esperando a que otro hilo realice una operación determinada.
* **TIME\_WAITING**: **Esperando un tiempo**. Esperando a que otro hilo realice una operación determinada en un periodo de tiempo concreto.
* **TERMINATED**: **Finalizado**. El hilo ha terminado su ejecución. (Normalmente finalizando su método run).

(WAITING Y TIME\_WAITING en los apuntes parecen mostrarse como NO\_RUNABBLE. Referente a que hay alguna actividad interna al propio hilo que se lo impide, como una espera por una operación de E/S. No se les puede asignar tiempo de procesamiento).

El método **getState** devuelve el estado en el que se encuentra el hilo. Todos los estados pasan por NEW, RUNNABLE y TERMINATED. El resto están condicionados a circunstancias propias de la ejecución.





**Iniciar un hilo**

Observamos que cuando se crea un hilo está en estado NEW. Pasa a RUNNABLE cuando se llama al método start(). Este método:

* Crea los recursos del sistema necesarios para ejecutar el hilo.
* Llama a su método run y lo ejecuta como un subproceso nuevo e independiente. (Si nosotros llamamos a run por nuestra cuenta no vamos a conseguir que esto suceda)
* Aunque se diga que el hilo está “corriendo” no tiene porqué estar ejecutándose. Podría simplemente estar preparado, esperando a tener tiempo de procesamiento.
* La ejecución de los hilos se entremezcla
* El orden de ejecución de los hilos no es determinístico.
  + El orden de ejecución de los hilos no tiene por qué coincidir con el orden en el que se iniciaron desde el hilo principal (main), o sea, no se conoce la secuencia con la que se ejecutan las instrucciones del programa.
  + En cada ejecución la mezcla de los hilos será diferente.

**Detener temporalmente un hilo**

Cuando un hilo está temporalmente detenido (no ejecutable) puede ser por:

* Hilo dormido. Se invocó al método sleep de la clase Thread durante un tiempo determinado. Después de ese tiempo vuelve a ejecutable.

El método **sleep** puede arrojar una *InterruptedException* (necesita de try-catch para controlarlo) y funciona introduciendo el tiempo en milisegundos sleep(milisegundos) o también con un tiempo adicional en nanosegundos sleep(milisegundos, nanosegundos).

Es necesario si queremos construir un reloj que nos marque cada segundo a su tiempo, si queremos visualizar números en un marcador gráfico, etc…

* Hilo esperando. Se ha tenido la ejecución mediante llamada al método **wait**. Volverá a ser ejecutable cuando se llame al método **notify** o **notifyAll** por otro hilo que ocupe el mismo proceso.
* Hilo bloqueado. Está pendiente de que finalice una operación de E/S en un dispositivo o a la espera de otro recurso. Ha sido bloqueado por el sistema operativo. Al finalizar la operación, vuelve a ejecutable.

Métodos que están deprecated (en desuso porque no son seguros y pueden provocar problemas)

* Método suspend. Detener temporalmente un hilo
* Método resume. Reanudar un hilo.
* Método stop. Para finalizar un hilo.

**Finalizar un hilo**

* El hilo finaliza cuando termina de ejecutarse su método run.
* Una vez que ha finalizado, no es posible volver a ejecutarlo con start. Debe crearse un nuevo hilo con new e iniciar el hilo con start.
* El hilo está vivo desde la llamada al método START hasta su muerte en FINISHED (Es decir, mientras está en los estados RUNNABLE, WAITING, BLOCKED y TIME\_WAITING) y no-vivo cuando está en los estados (NEW o FINISHED). Con el método **isAlive** se puede consultar si está vivo o no.
* Con el método join podemos esperar hasta la muerte del hilo.

Iniciamos el hilo: hiloAuxiliar.start()

Y si después ponemos: **hiloAuxiliar.join()**

Esperará hasta que acabe el hilo para ejecutar el resto de instrucciones del programa principal. (Arroja excepción, debo usar try-catch)

Texto

Descripción generada automáticamente

Cuando ejecutamos una aplicación de ventana basada en un formulario JFrame, la máquina virtual de Java crea tres hilos por defecto:

1. **el principal, donde corre el método Main()** que inicia la aplicación
2. **el conocido como GC (recolector de basura**), donde corre el código que se encarga de optimizar los recursos de memoria
3. **el conocido como EDT (hilo despachador de eventos) de la AWT (paquete abstracto de herramientas de ventana),** donde corre el código encargado de atender las solicitudes de dibujo del contenido de la ventana, y de atender a los eventos de ratón o teclado provocados por el usuario (como la ejecución del código programado para cuando el usuario hace click en un botón)

*Puesto que el hilo EDT puede llegar a ser el más ocupado de los tres con diferencia, debe de ponerse especial cuidado en no sobrecargarlo demasiado.*

En esta aplicación queremos realizar una cuenta de 1 a 20, cuyo avance se vaya reflejando en el JPanel que hemos colocado a modo de marcador. La cuenta se pondrá en marcha cuando el usuario haga click en un botón.

Obviamente, para reflejar ese avance es necesario repintar el JPanel a cada nuevo valor (lo que se le solicita al hilo EDT -que es el encargado de ello, mediante una llamada al método repaint() del JPanel).

* El primer botón intenta hacerlo sin poner más hilos en juego. No lo consigue. Y no lo consigue porque precisamente el hilo que recibe las peticiones de pintado, es justo el que las está enviando a cada nuevo valor de la cuenta. Y claro, se satura. Sólo cuando finaliza la cuenta (justo en el 20), puede atender por fin a la solicitud de dibujo del marcador.
* Por el contrario el segundo y tercer botón delega en un hilo auxiliar, tanto la cuenta como las peticiones de repintado del marcador.
* Además este hilo auxiliar, en el caso del Botón 3 se duerme 1 décima de segundo por cuenta, para que EDT tenga una buen ocasión de atender la petición de dibujo que le acaba de enviar. El resultado es el esperado por fin.

1. **Gestión y planificación de hilos**

Como ya se ha hablado, la ejecución de hilos se puede realizar mediante:

* Paralelismo: Sistema con múltiples CPU o núcleos en los que cada CPU ejecuta un hilo diferente.
* Pseudoparalelismo: Una CPU responsable de ejecutar múltiples hilos, lo que requiere de planificar una secuencia de ejecución (sheduling)

El planificador de hilos de Java (Sheduler) usa un algoritmo de secuenciación de hilos (fixed priority sheduling), sistema de prioridades relativas, en la que la ejecución de hilos se hace en base a las prioridades de cada uno.

Funciona de la siguiente forma:

* El hilo elegido para ejecutarse siempre es el “RUNNABLE” de prioridad más alta.
* Si hay más de un hilo con misma prioridad se hace algoritmo por turnos (round-robin) basado en una cola circular FIFO (primero en entrar, primero en salir).
* Cuando el hilo en ejecución pasa a estados no ejecutables o a estado muerto, se selecciona otro hilo para la ejecución.
* La ejecución el hilo se interrumpe si otro hilo de prioridad más alta se vuelve RUNNABLE. A esto se le llama “planificación apropiada” (preemptive sheduling)

La responsabilidad de ejecución de los hilos es del sistema operativo sobre el que corre la JVM pudiendo ser de dos formas:

* Sistema operativo que implementa time-slicing (subdivisión del tiempo): El hilo que entra en ejecución se mantiene en la CPU solo un microintervalo de tiempo fijo (quantum) de procesamiento. El hilo en ejecución, además de interrumpirse cuando uno de prioridad más alta se vuelve ejecutable, también lo hace cuando su quantum de ejecución se acaba. (Linux y Windows siguen este patrón)
* 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 por Solaris.

Tabla

Descripción generada automáticamente con confianza media

* 1. **Prioridad de hilos**

En Java cada hilo tiene una prioridad entre 1 y 10 (Más valor, más prioridad).

El hilo principal (el que ejecuta el método main) siempre tiene prioridad 5.

Los hilos secundarios heredan la prioridad que tenga en ese momento su hilo padre.

La prioridad de un hilo se puede obtener y modificar con getPriority() y setPriority(). Los niveles de prioridad de Java no tienen por qué coincidir con los del sistema operativo. Por eso, se recomienda usar en el código las constantes de prioridades ya disponibles en la clase Thread (asignando prioridad alta para tareas de tiempo crítico y baja para tareas menos importantes). Las constantes son:

* MAX\_PRIORITY (=10)
* MIN\_PRIORITY (=1)
* NORM\_PRIORITY (=5)

Los hilos demonio en java (son hilos que pueden ejecutarse en segundo plano dando servicio a otros hilos). Java los usa para subprocesos de usuario y de recolección de basura, etc. Los hilos demonio se crean con el método **setDaemon(boolean)** (debe introducirse true o false). El método **isDaemon()** pueden comprobar si el hilo en ejecución es un hilo demonio o no.

**Cualquier hilo que cree un hilo demonio, será automáticamente también un hilo demonio. Como los hilos demonio dan servicio a los hilos de usuario, cuando estos finalicen, el intérprete de java también finaliza; sin esperar a que los hilo demonio lleguen a su final.**

Texto

Descripción generada automáticamente

* 1. **Hilos egoístas y programación expulsora**

En los sistemas operativos sin time-slicing puede suceder que el hilo que entra en ejecución no saga de ella hasta terminar; sin dejar que los hilos preparados entre en ejecución.

Para evitar el comportamiento egoísta de uno de los hilos, existe el método de programación expulsora **yield()** y que permite que hilos de otros hilos de la misma prioridad puedan ejecutarse.

Deben incluirse llamadas a yield en la aplicación, incluso a sleep o a wait si el hilo no se bloquea por una E/S.

(si la invocación se hace desde hilo Runnable, debe ser Thread.yield() como ya vimos anteriormente).

El funcionamiento de yield no está garantizado. Puede que el hilo egoísta que pasa a “preparado” vuelva a ejecutarse.

1. **Sincronización y comunicación de hilos**

Los hilos pueden necesitar compartir recursos o información si:

* Dos hilos compiten por obtener un mismo recurso (acceder a un mismo archivo o variable)
* Dos hilos colaboran para obtener un fin común y para ello deben comunicarse a travé de un recurso (un hilo produce información que utilizará otro hilo).

Esto se realiza mediante la sincronización y comunicación de hilos.

SINCRONIZACIÓN: Capacidad de informar de la situación de un hilo a otro (para establecer secuencialidad correcta del programa).

COMUNICACIÓN: Capacidad de transmitir información de un hilo a otro.

La sincronización y comunicación se consigue en Java mediante:

* Monitores: Marcar bloques de código con **synchronized**.
* Semáforos: Variable especial (o tipo abstracto de datos) que permite restringir el acceso a recursos compartidos. Implementar propios semáforos o usar Semaphore. Semáforo binario, indicador de condición que registra si un recurso está disponible o no.
* Notificaciones: Comunicar hilos mediante **wait, notify y notifyAll.**
* Clases de sincronización: **Semaphore, CountDownLatch, CyclicBarrier y Exchanger.**

Sección crítica: Sección de código que no puede ejecutarse concurrentemente porque se encuentran recursos compartidos por diferentes hilos.

Condición de carrera: Cuando varios hilos acceden a la vez a un mismo recurso (variable) cambiando su valor y obteniendo un valor no esperado de la misma.

**Las secciones críticas se protegen mediante sincronización:**

* 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 no se cumpla una determina condición.

Texto

Descripción generada automáticamente

* 1. **Monitores (Synchronized)**

La sincronización entre recursos compartidos puede realizarse en Java implementando **MONITORES**.

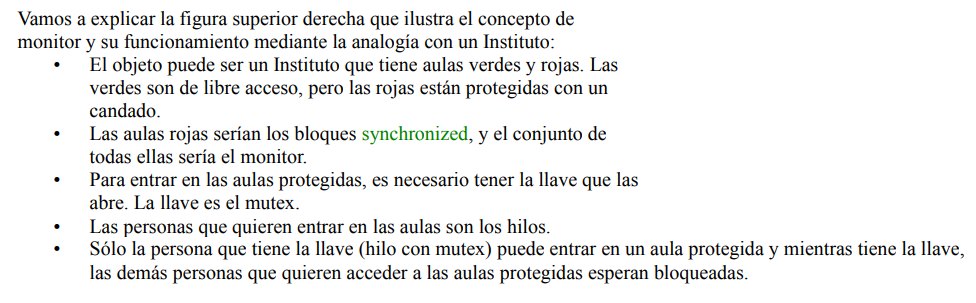
Un monitor es una porción de código protegida por un mutex o lock (elemento que protege recursos con exclusión mutua). En java se implementa marcando el segmento de código con la palabra **synchronized**. Interesa marcar los bloques que se correspondan con secciones críticas y contengan el código o datos que comparten los hilos.

Texto

Descripción generada automáticamente con confianza baja

Los monitores funcionan de la siguiente forma:

* **Están asociados a un objeto específico y solo a un monitor por objeto (aunque este tenga más de un bloque synchronized)**
* **Solo un hilo puede tener el candado de un objeto en un momento dado.**
* El reto de hilos que también necesitan hacerse con el mutex de ese objeto para acceder a los bloques synchronized, permanecen bloqueados a la espera de que se libere el mutex del objeto.
* Cuando el hilo finaliza la ejecución de un bloque synchronized, libera el mutex.
* Al liberarse el mutex, todo lo hilos en espera se reactivan y el planificador de Java cede el mutex a uno de ellos
* **El monitor solo permite que un hilo ejecute un bloque synchronized a la vez**. Por lo que **si hay varios bloques synchronized dentro de un objeto, solo uno de ellos podrá ejecutarse al mismo tiempo**.
* Imagen que contiene Diagrama

  Descripción generada automáticamenteLos monitores java son reentrantes, esto es, un mismo hilo puede adquirir el mutex varias veces (el hilo no se excluye a sí mismo; permitiendo que varios métodos synchronized se llamen entre sí, incluso recursivamente).

El hilo que entra en un método synchronized se puede encontrar con dos situaciones:

* Mutex libre. El hilo toma el mutex, ejecuta el método y lo libera cuando haya finalizado la ejecución.
* Mutex en posesión de otro hilo. El hilo se bloquea en espera de que el primero lo libere.

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

Interfaz de usuario gráfica, Texto, Aplicación

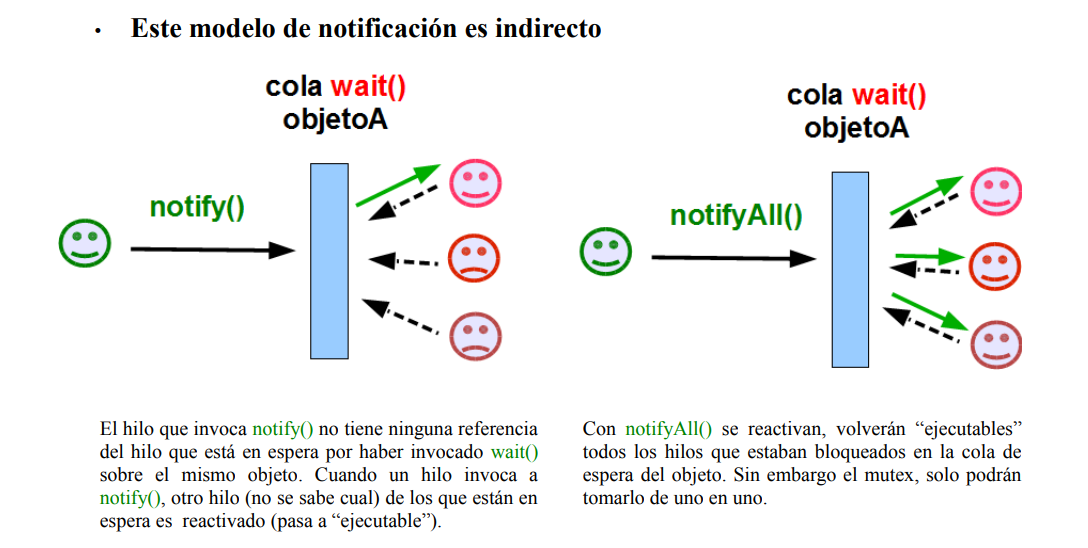
Descripción generada automáticamente

* 1. **Comunicación entre hilos**

Los métodos wait, notify y notifyAll permiten sincronizar los accesos al monitor.

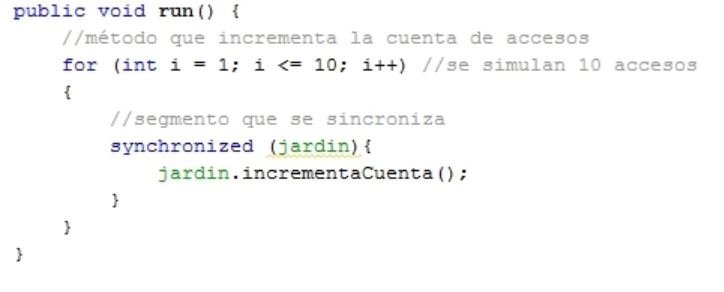
Supone un mecanismo de autosincronización para lograr que un hilo actúe solo cuando ha concluido cierta actividad.

* **wait()** Detiene al hilo que lo invoca (pasa a no ejecutable) y no se reanuda hasta que otro hilo notifique la posibilidad de continuar. Para poder invocarlo, debe tener el mutex del objeto compartido. Al detenerlo, lo pone en espera y libera el mutex del objeto.
* **wait(tiempo)** Como wait() pero el hilo también se reanuda si pasa el tiempo esperado
* **notify()** Notifica a uno de los hilos que invocó wait() para el mismo objeto que ya puede continuar. Reobtiene el mutex del objeto.
* **notifyAll()** Notifica a todos los hilos puestos en espera para el mismo objeto que ya pueden continuar.



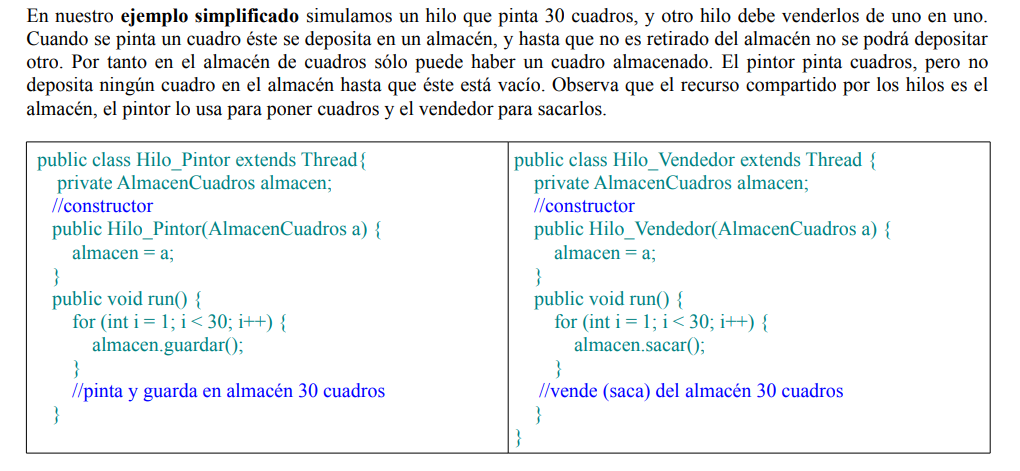
La llamada a estos métodos se hace dentro de bloques syncronized.

A veces no vamos a poder sincronizar un método. No queda otra que poner las llamadas a los métodos que se quieren sincronizar dentro de segmentos sincronizados como:

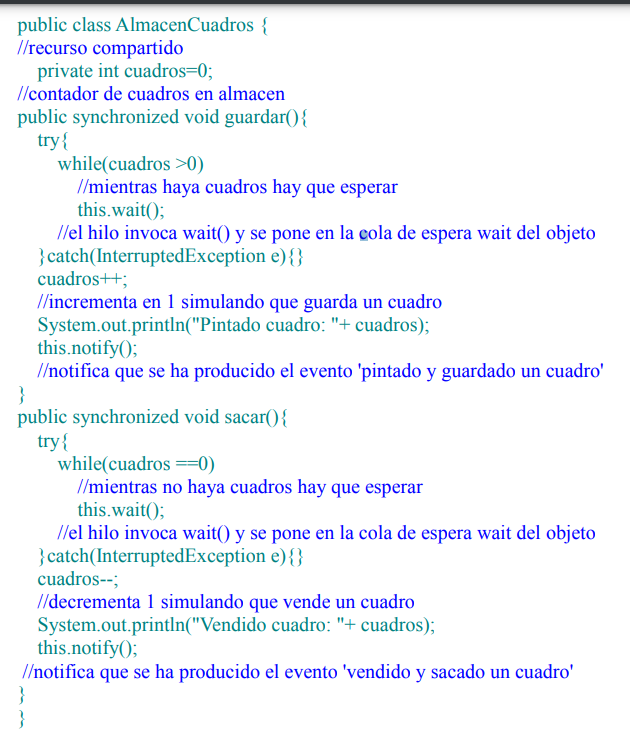


Cuidado: Declarar un segmento de código como sincronizado ralentiza la ejecución del programa (la adquisición y liberación de monitores genera sobrecarga). Siempre que sea posible es mejor sincronizar método completos. Al declarar estos bloques, puede aparecer interbloqueo.

* + 1. **Problemas relacionados con la comunicación entre hilos**

Hay dos problemas clásicos sobre la necesidad de sincronizar y comunicar hilos:

**Productor-Consumidor.** Modelar el acceso simultáneo de varios hilos a una estructura de datos u otro recurso, de forma que unos hilos almacenan los datos en el recurso y otros se encargan de eliminar y procesar esos datos.



Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

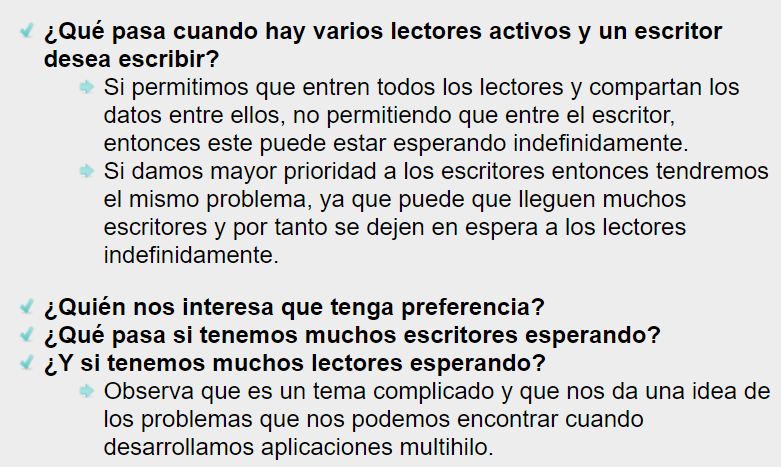
* **Lectores-Escritores**. Modelas el acceso simultáneo de varios hilos a base de datos o fichero, queriendo uno leer y otros escribir o modificar.

Varios lectores pueden leer a la vez; pero cuando un escritor accede no puede haber ningún lector, ni escritor usándola.

Puede hacerse uso de la clase Semáforo (creada por nosotros) que indica el estado en el que se encuentra el recurso (LIBRE, CON\_LECTORES, CON\_ESCRITOR)

Si, según el semáforo, hay un escritor, el resto queda en wait()

Cuando finaliza la escritura se notifica notify() que la base de datos ha quedado libre. Cuando finaliza la lectura (y no hay más lectores dentro), entonces se notifica que la base de datos ha quedado libre.



* + 1. **Más problemas relacionados con la comunicación entre hilos**

**El problema del interbloqueo (deadlock)**

Sucede cuando un hilo se bloquea y espera indefinidamente debido a que:

* Un hilo espera que le llegue aviso de otro que nunca llega
* Todos los hilos, de forma circular, esperan para acceder a un recurso.

Pasa cuando un hilo entra en un bloque synchronized y a su vez llama a otro bloque synchronized o bien cuando se usan clases de java.util.Concurrent que llevan implícita la exclusión mutua. Algunos problemas de interbloqueo se resuelven con la interface Lock.

**El problema de la inanición (starvation).**

Un hilo es desestimado para su ejecución. No puede tener acceso regular a los recursos compartido y no puede avanzar quedándose bloqueado.

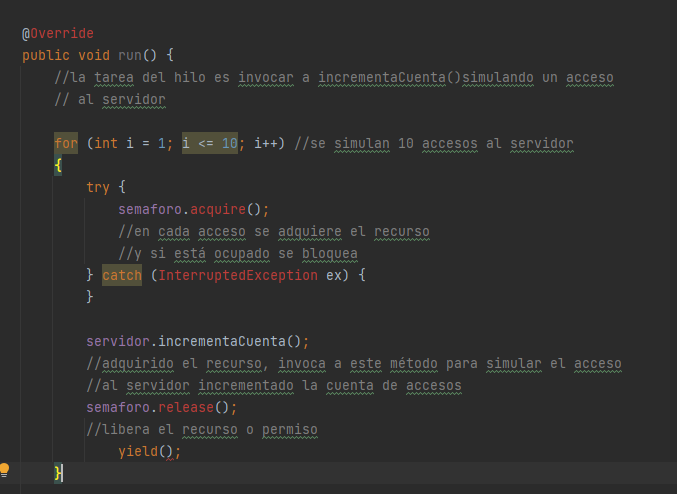
[Otros problemas: Condiciones deslizadas (La condición de evaluación cambia después de la evaluación y antes de la ejecución), inconsistencia de memoria (simultáneamente valores diferentes)]

* 1. **La clase Semaphore**

Permite definir un semáforo para controlar el acceso a un recurso compartido.

**Crear y usar el objeto Semaphore**

* Indicar al constructor *Semaphore(int permisos)* el número de permisos que se pueden dar para acceder al recurso compartido. Coincide con el número de hilos que pueden acceder a la vez al recurso.
* Indicar al semáforo mediante método **adquire()** que se quiere acceder al recurso o mediante **adquire(int permisosAdquirir)** el número de permisos que se quieren consumir.
* Indicar al semáforo mediante **release()** que libere el permiso o mediante **release(int permisosLiberar)** el número de permisos que se quieren liberar.
* Usar el constructor Semaphore(int permisos, boolean justo) que garantiza que el primer hilo en invocar **adquire()** será el primero en adquirir un permiso cuando se libere. (Garantiza el orden de adquisición de permisos)
* Si se usa Semaphore para proteger secciones críticas, debe llamarse a **adquire()** y **release()** desde la sección crítica y dar solo 1 permiso en el contructor.
* Si se usa para comunicar hilos, un hilo debe invocar a **adquire()** y el otro a **release()** para trabajar de manera coordinada y debe darse un número de permisos correspondiente al máximo número de hilos bloqueado en la cola para adquirir permisos.



Texto

Descripción generada automáticamente

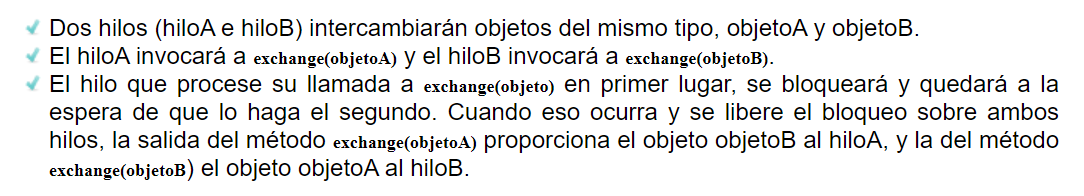
Texto

Descripción generada automáticamente

* 1. **La clase Exchanger**

Establece un punto de sincronización **donde se intercambian objetos entre dos hilos.**

Exchanger<V> es genérica (en <V> debe especificarse el tipo de objeto a compartir. El hilo que desea obtener la información espera realizando una llamada al método **exchange()** hasta que otro hilo sitúe la información usando el mismo método o hasta que pase un periodo de tiempo establecido mediante el parámetro timeout.



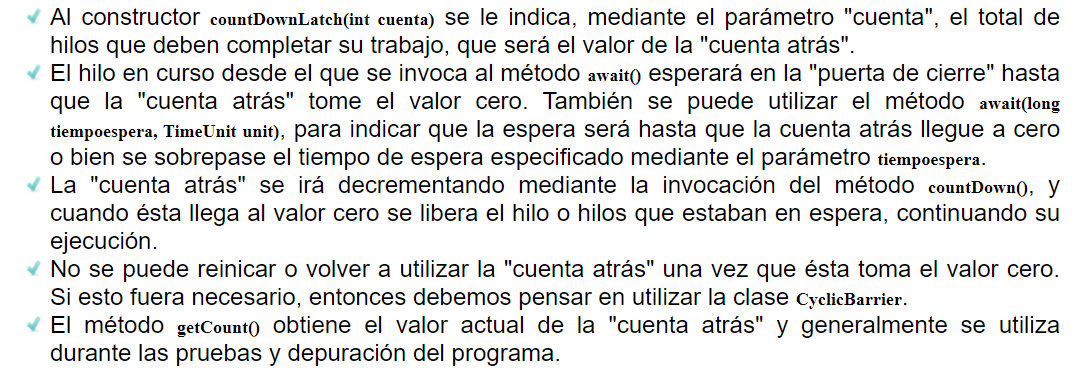
Texto

Descripción generada automáticamente

* 1. **La clase CountDownLatch**

Permite que uno o más hilos esperen hasta que otros hilos finalicen su trabajo. “Cuenta atrás de cierre”.

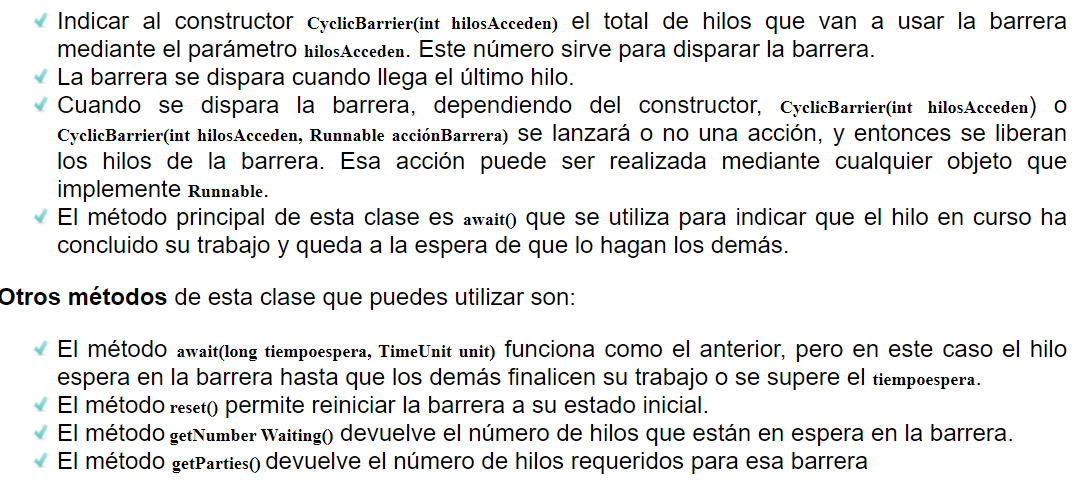
Implementa un punto de espera “puerta de cierre” donde uno o más hilos esperan a que otros finalicen su trabajo. Los que deben finalizar su trabajo se controlan mediante un contador (cuenta atrás). Cuando la “cuenta atrás” llega a cero se reanuda el trabajo de los hilos interrumpidos y puestos en espera.



* 1. **La clase CyclicBarrier**

Permite que uno o más hilo esperen hasta que todos ellos finalicen su trabajo.

Se implementa un punto de espera “barrera” donde cierto número de hilos esperan hasta que todo ellos finalicen el trabajo. Finalizado el trabajo de los hilos, se dispara la ejecución de una acción determinada o el hilo interrumpido continúa su trabajo. La barrera se llama cíclica porque se puede volver a usar después de que los hilos han espera han sido liberados tras finalizar todos su trabajo y también se puede volver a utilizar.



1. **Utilidades de concurrencia**
   1. **Algunas clases más de Java**

* **Interfaz Executor (Programador de tareas)**

Permite ejecutar tareas en un único hilo en segundo plano, en hilo nuevo o en pool e hilos. Ejecutar tareas mediante execute() (estas deben implementar la interfaz Runnable). Usar implementaciones de Executor como ExecutorService.

* **Colecciones**
  + Interfaz **Queque**. Almacenar elementos antes de procesarlos (con operaciones de inserción, extracción e inspección)
  + Interfaz **BlockingQueque**. Diseñada para colas de tipo productor/consumidor, son thread-safe. Esperan mientras no haya elementos almacenados en la cola.
  + Implementaciones concurrentes de Map y List. Seguras en entornos multihilo porque equivalen a otras estructuras sincronizadas: **ConcurrentHashMap** (HashMap), **ConcurrentSkipListMap** (TreeMap), **CopyOnWriteArrayList** (ArrayList), **CopyOnWriteArraySet** (Set)
* **Clase Locks**
  + Implementaciones y bloqueos y desbloqueos entre métodos. Es equivalente a Synchronized pero dando métodos que hacen más fácil el uso de bloqueos y condiciones. Algunos de ellos son:
    - newCondition(), que genera Condition asociada al bloqueo.
    - await(), que indica cuándo deseamos esperar
    - signal(), que indica si una condición de bloqueo se activa para finalizar la espera
    - Implementación ReentranLock permite realizar exclusión mutua usando monitores. lock() indica que deseamos usar el recurso compartido y unlock() que hemos acabado de usarlo.
* **Variables atómicas** 
  + Definir recursos compartidos sin necesidad de protegerlos ya que ellas internamente realizan dichas labore de protección.
  1. **Los pool de hilos**

Al trabajar con aplicaciones de tipo servidor, estas tienen que atender un número masivo y concurrente de peticiones de usuario en forma de tareas que deben ser procesadas lo antes posible. Es conveniente usar hilos, pero si se ejecuta cada tarea en un hilo distinto se pueden llegar a crear tantos que puede acabar comprometida la estabilidad del sistema. Para evitar esto están los pools de hilos.

Un pool de hilos es un contenedor dentro del cual se crean e inician un número limitados de hilos para ejecutar todas las tareas de una lista.

Los pool se declaran como objetos **ExecutorService** (que es una implementación de la interfaz Executor) usando los métodos:

* **newFixedThreadPool(int numeroHilos)** : Crea pool con el número de hilos indicado. Estos hilos se reutilizan cíclicamente hasta terminar con las tareas de la cola o lista.
* **newCachedThreadPool()** : Crea pool que va creando hilos conforme se va necesitando pero que puede reutilizar los ya concluidos para no tener que crear demasiados. Los hilos inactivos durante un tiempo son finalizados automáticamente por el pool.
* **newSingleThreadExecutor()** : Crea un pool de un solo hilo. Si ocurre excepción en la ejecución de una tarea, no se detiene la ejecución de las siguientes.
* **newSheduledExecutor()** : Crea un pool que va a ejecutar tareas programadas cada cierto tiempo, ya sea de una sola vez o de forma repetitiva. Es parecido a un Timer pero con la diferencia de que puede crear varios hilos que irán realizando tareas programadas conforme se desocupen.

Debe llamarse al método **execute()** una vez por cada tarea que deba ser ejecutada en el pool (se pasa como argumento del método).

Hay métodos para el control de la ejecución de las tareas como el método **shutdown()** para indicarle al pool que los hilos no se reutilizaran y que deben morir cuando finalicen su trabajo.

* 1. **Control de excepciones**

Texto

Descripción generada automáticamenteLa excepciones de una aplicación multihilo pueden controlarse usando el método **setUncaughtExceptionHandler(manejador)** que permite definir un manejador de excepciones para un hilo determinado.

¿Cómo se construye un manejador?

* Se crea una clase que implemente la interfaz **Thread.UncaughtExceptionHandler**
* Se implementa el método **uncaughtException()** con lo que queremos hacer cuando se produzca una excepción en el hilo manejado por el manejador.
* Interfaz de usuario gráfica, Texto, Aplicación

  Descripción generada automáticamente

**7.4. Depuración de aplicaciones multihilo**

* La depuración de aplicaciones multihilo es difícil debido a que:
  + La ejecución de hilos tiene comportamiento no determinístico
  + Hay que controlar varios flujo de ejecución
  + Hay errores potenciales debido a la compartición de recursos entre hilos
  + Hay errores por no cumplirse la exclusión mutua
  + Hay errores por interbloqueo

El seguimiento puede realizarse con métodos como:

**dumpStack()** Traza de la pila del hilo en curso

**getAllStackTrace()** Map con todos los hilos vivos en la aplicación

**getStackTrace()** Seguimiento de la pila de un hilo

Estos dos últimos permiten grabar los datos de seguimiento de la pila en un log.

1. **Aplicaciones multihilo**

**Propiedades de una aplicación multihilo**

* Seguridad: Evitar estados inconsistentes *por mal uso de los recursos compartidos*. Se sincronizan los hilos asegurando exclusión mutua.
* Viveza: *Evitar bloqueos o que un hilo no pueda ejecutarse*. Comportamiento no egoísta de los hilos y ausencia de interbloqueos e inanición.

**Corrección de la aplicación**

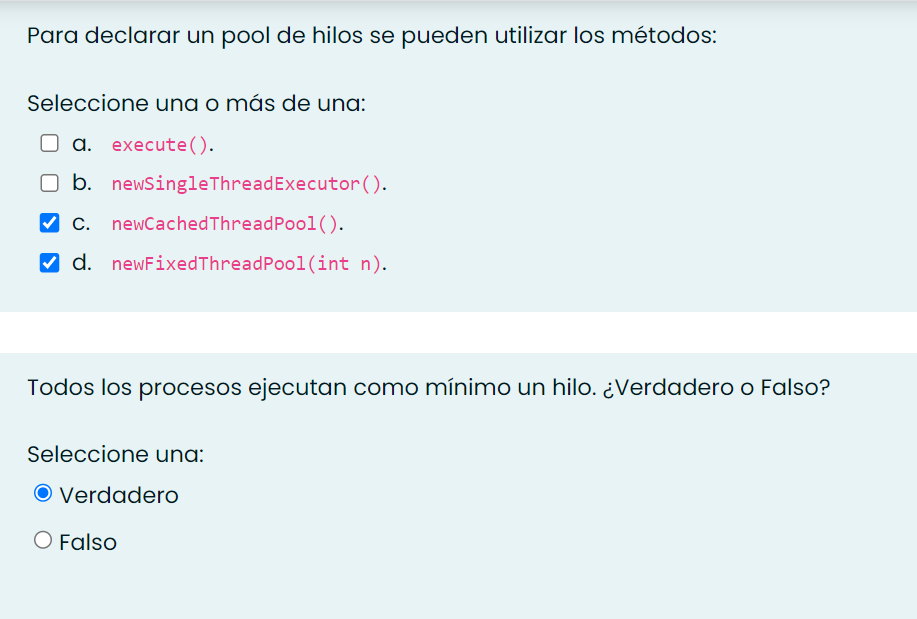
* Corrección parcial: Se cumple la propiedad de seguridad.
* Corrección total: Se cumple la propiedad de seguridad y viveza.

**Aspectos para desarrollar aplicaciones multihilo**

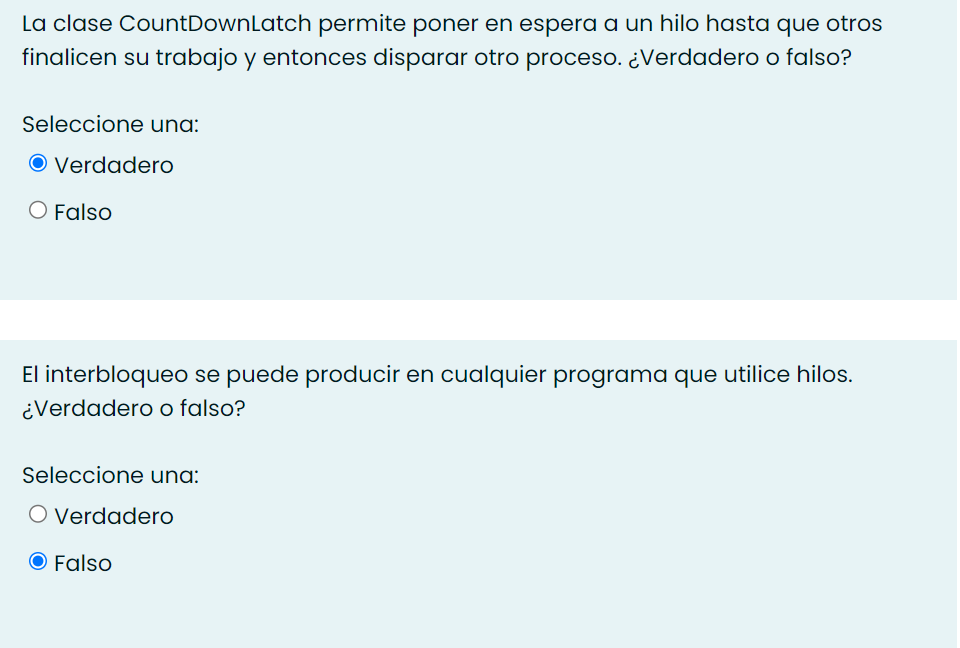
* Situación en los hilos en la aplicación:
  + Independientes: No es necesario sincronizar y/o comunicar hilos
  + Colaborando / Compitiendo: Debe sincronizarse y/o comunicar hilos, evitando interbloqueos y esperas indefinidas.
* Gestión de prioridades: Hilos más importantes deben ejecutarse antes.
* Considerar la posibilidad de que el SO no implemente time-slicing.
* Considerar que la ejecución de hilos es no-determinística.

**Ventajas del uso de librerías:**

* Facilitar la programación
* Mayor rendimiento
* Mayor fiabilidad
* Menor mantenimiento
* Mayor productividad









Interfaz de usuario gráfica, Texto, Aplicación, Correo electrónico

Descripción generada automáticamente



Interfaz de usuario gráfica, Texto, Aplicación, Correo electrónico

Descripción generada automáticamente

Interfaz de usuario gráfica, Texto, Aplicación, Correo electrónico

Descripción generada automáticamente