Programación multihilo.

# 1.- Introducción.

La programación multihilo es otro tipo de programación concurrente.

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**. Hace las tareas una detrás de otra, de manera secuencial. Cada una de ellas debe concluir por completo, antes de iniciar la siguiente
* **Programa de flujo múltiple**. Reparte las actividades 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**.

El que las tareas se ejecuten realmente en paralelo **dependerá** del **S.O** y del **nº de procesadores** del sistema donde se ejecute la aplicación. En realidad, lo importante es la **sensación real** de que el programa realiza de forma **simultánea** diferentes tareas.

**Para saber más**: [historia de la programación multihilo, así como alguna de sus aplicaciones](http://resenadelossistemasoperativos.blogspot.com/2013/09/hilos-en-sistemas-operativos.html).

Interfaz de usuario gráfica, Texto

Descripción generada automáticamente

# 2.- Conceptos sobre hilos.

**Caso practico**: la programación multihilo es una técnica de programación muy poderosa, pero también peligrosa.

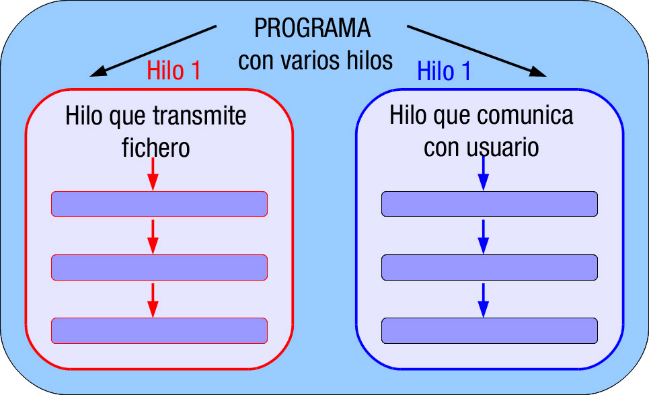
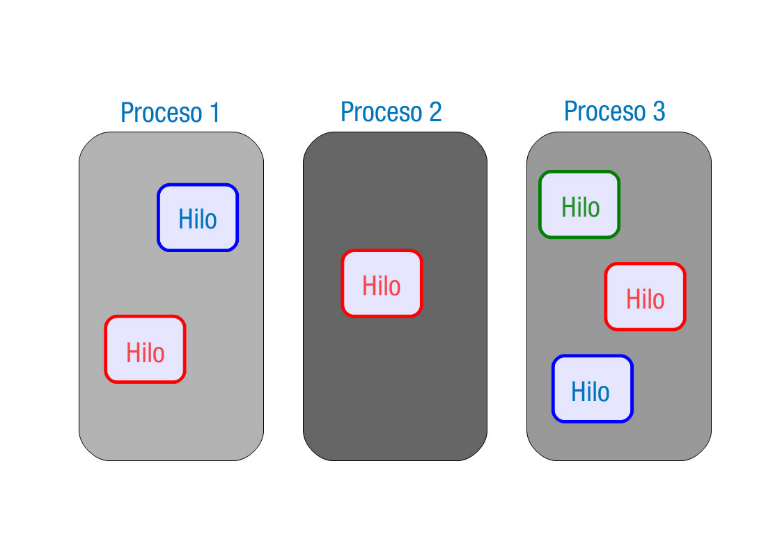
Un **hilo**, denominado también **subproceso**, es un **flujo de control secuencial independiente dentro de un proceso** y está asociado con una secuencia de instrucciones, un conjunto de registros y una pila.

Cuando se ejecuta un programa, el Sistema Operativo crea un proceso y también crea su primer hilo, **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**.

**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**.



Un único hilo es similar a un programa secuencial. Sin embargo, ejecutar varios hilos dentro de un proceso ofrece algo nuevo y útil: hacer cada uno de ellos una actividad diferente al mismo tiempo.

**Para saber más**: [Hilo (informática) - Wikipedia, la enciclopedia libre](https://es.wikipedia.org/wiki/Hilo_(inform%C3%A1tica))

Texto

Descripción generada automáticamente

## 2.1.- Recursos compartidos por los hilos.

Un hilo se compone de:

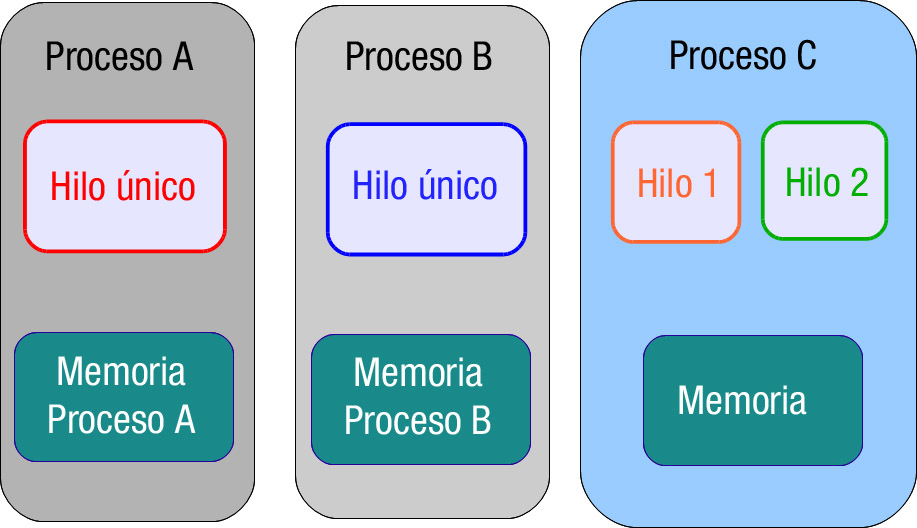
* Un identificador único.
* Un contador de programa propio.
* Un conjunto de registros.
* Una pila (variables locales).

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.

Por tanto, **si un hilo corrompe la memoria del proceso que comparten, los demás hilos también se verán afectados**. En los procesos esto no ocurría, porque el S.O protegía el espacio de memoria de cada proceso.

El hecho de que los hilos compartan recursos (por ejemplo, pudiendo acceder a las mismas variables) implica que sea necesario **utilizar esquemas de bloqueo y sincronización**, lo que puede hacer más difícil el desarrollo de los programas y 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 programas que se bloqueen con facilidad y/o que intercambien datos de manera equivocada.

Texto

Descripción generada automáticamente

## 2.2.- Ventajas y uso de hilos.

**Ventajas**:

* Se consumen menos recursos en el lanzamiento, y la ejecución de un hilo que en el lanzamiento y ejecución de 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**? Se aconseja utilizar hilos en una aplicación cuando:

* La aplicación **maneja entradas de varios dispositivos** de comunicación.
* La aplicación debe poder **realizar diferentes tareas a la vez**.
* **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**.

**Por ejemplo**: los servidores web y de bases de datos tienen un hilo principal, que queda a la escucha de peticiones por parte de clientes. Cuando recibe una nueva petición, crea un nuevo hilo que se encarga de procesarla y generar la consulta, mientras que el hilo principal sigue a la escucha recibiendo peticiones y creando hilos.

Sin embargo, **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 gráfico 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.

Texto

Descripción generada automáticamente

# 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 **java.lang** y con método específicos para la manipulación de hilos en la clase **Object**.

A partir de la versión 5, en Java SE se incluye el paquete **java.util.concurrent** con nuevas utilidades para desarrollar aplicaciones multihilo e incluso aplicaciones con un alto nivel de concurrencia.

**Para saber más**: [Processes and Threads (The Java™ Tutorials > Essential Java Classes > Concurrency) (oracle.com)](https://docs.oracle.com/javase/tutorial/essential/concurrency/procthread.html), [Multitarea e Hilos en Java con ejemplos (Thread & Runnable) - Jarroba](https://jarroba.com/multitarea-e-hilos-en-java-con-ejemplos-thread-runnable/)

## 3.1.- Utilidades de concurrencia del paquete java.lang.

* **Clase thread**. Es la clase 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.
* **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**().

En el siguiente enlace, hay una **tabla resumida de la clase *thread***, que incluye los métodos más comunes para gestionar y controlar hilos.

[Resumen uso de la clase ***Thread***](https://aula21-my.sharepoint.com/personal/5634224_alu365_murciaeduca_es/Documents/Documentos/FP/DAM/Curso%202/Materias/Programación%20de%20servicios%20y%20procesos/Evaluacion%201/Tema%202/Recursos/tema3_2_concurrencia2.pdf).

**Para saber más**: [métodos incluidos en la clase Thread](https://docs.oracle.com/en/java/javase/13/docs/api/java.base/java/lang/Thread.html).

Texto

Descripción generada automáticamente

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

Este paquete **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.
    - Exchanger.
  + **Interfaces para separar la lógica de la ejecución**, como, por ejemplo:
    - Executor.
    - ExecutorService.
    - Callable.
    - Future.
  + **Interfaces para gestionar colas de hilos**.
    - BlockingQueue.
    - LinkedBlockingQueue.
    - ArrayBlockingQueue.
    - SynchronousQueue.
    - PriorityBlockingQueue.
    - DelayQueue.
* **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.
  + AtomicLong.
* **java.util.concurrent.locks**. Define una serie de clases **como uso alternativo a la cláusula sinchronized**. En este paquete se encuentran algunas interfaces como, por ejemplo:
  + Lock.
  + ReadWriteLock.

**Para saber más**: [java.util.concurrent (Java SE 13 & JDK 13 ) (oracle.com)](https://docs.oracle.com/en/java/javase/13/docs/api/java.base/java/util/concurrent/package-summary.html)

# 4.- Creación de hilos.

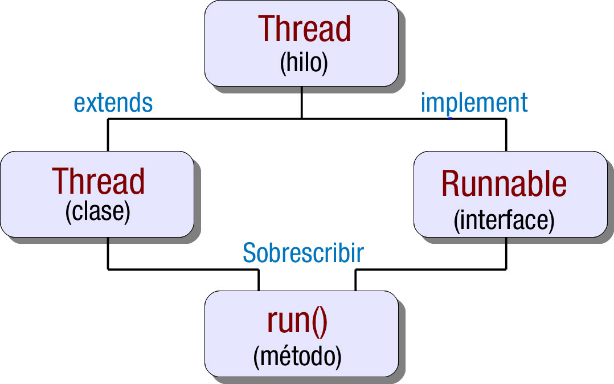
En Java, un **hilo** se representa mediante una instancia de la clase **java.lang.thread**. Este **objeto** **thread** se emplea para **iniciar, detener o cancelar la ejecución** del hilo de ejecución.

**Cómo se definen los hilos o threads**:

* Heredando de la clase ***thread***.
* Implementando 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

**¿Cuándo usar un procedimiento u otro?**

* **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.
* **Implementar Runnable siempre es posible**, es el procedimiento más general y también el más flexible.

Durante la ejecución de un programa, controlado por el método **main()**, se está ejecutando el hilo principal, y **es el último hilo que termina su ejecución**, ya que cuando este hilo finaliza, el programa termina.

[**Ejemplo práctico**](https://aula21-my.sharepoint.com/personal/5634224_alu365_murciaeduca_es/Documents/Documentos/FP/DAM/Curso%202/Materias/Programación%20de%20servicios%20y%20procesos/Evaluacion%201/Tema%202/Apuntes/PSP02/WEB/ZIP/PSP02_CO.ZIP): para saber qué hilo se está ejecutando en un momento dado, el hilo en curso, utilizamos el método **currentThread()** y que obtenemos su nombre invocando al método **getName()**, ambos de la clase **thread**.

Texto

Descripción generada automáticamente con confianza media

## 4.1.- Creación de hilos extendiendo la clase Thread.

public class Saludo extends Thread {

//clase que extiende a Thread

public void run() {

// se redefine el metodo run() con el codigo 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

}

}

[Enlace al proyecto](https://aula21-my.sharepoint.com/personal/5634224_alu365_murciaeduca_es/Documents/Documentos/FP/DAM/Curso%202/Materias/Programación%20de%20servicios%20y%20procesos/Evaluacion%201/Tema%202/Apuntes/PSP02/WEB/ZIP/PSP02_-2.ZIP).

Texto

Descripción generada automáticamente

## 4.2.- Creación de hilos mediante la interfaz Runnable.

Para **definir y crear hilos implementando la interfaz *Runnable***, seguiremos los siguientes pasos:

* **Declarar una nueva clase que implemente a Runnable**.
* **Redefinir** en la nueva clase **el método run()** con el código asociado al hilo.
  + Aquí se pone **lo que queremos que haga el hilo**.
* **Crear un objeto de la nueva clase**.
* **Crear un objeto de la clase thread pasando como argumento** el objeto cuya clase tiene el método run(). Este será realmente el hilo.

Una vez creado el hilo, para ponerlo en marcha o iniciarlo:

* **Invocar al método start() del objeto thread** (el hilo que hemos creado).

**Ejemplo**: aquí se muestra cómo crear el hilo implementando ***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 metodo run() con el codigo 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 hilol= new Thread (miRunnable) ;

//se crea un objeto Thread (el hilo hilol) pasando como argumento

// al constructor un objeto Saludo

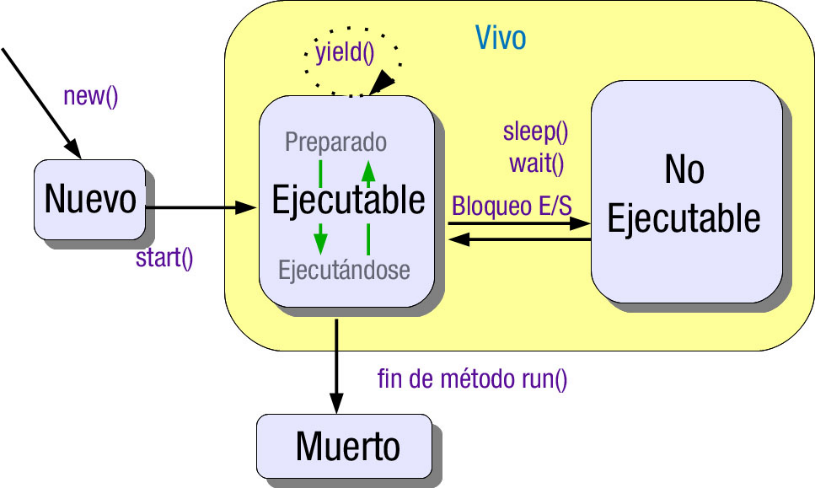
hilol.start ();

//se invoca al metodo start () del hilo hilo1

}

}

# 5.- Estados de un hilo.

**Caso práctico**: hay que tener muy claros los diferentes estados que puede tener un hilo a lo largo de su vida y cómo se pasa de uno a otro.

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.

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. En este estado, **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 está bloqueado por algún motivo y no tiene oportunidad de que se le asigne tiempo de procesamiento. **Por ejemplo**:
  + 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)**.
* **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**.

Texto

Descripción generada automáticamente

## 5.1.- Iniciar un hilo.

1. **Crear el hilo**: mediante el método new()
2. **Hacer que esté en estado “Ejecutable”**: método **start()** de la clase ***Thread***.

En realidad, 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.

**Consideraciones importantes**:

* **Invocar manualmente al método run()** ejecutará el código que hay en su interior, pero **no se ejecutará en un hilo**.
* **Llamar dos veces al método start()** provocará una excepción de tipo **IllegalThreadStateException**.
* **El orden de ejecución de los hilos es no-determinístico** (no se conoce la secuencia en la que serán ejecutadas la instrucciones del programa aunque lances los hilos en un determinado orden).

**Ejemplo**: programa que define dos hilos, construidos cada uno de ellos por los 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 (número de palabras que imprime cada hilo) para apreciar las observaciones indicadas anteriormente. [Enlace al proyecto](https://aula21-my.sharepoint.com/personal/5634224_alu365_murciaeduca_es/Documents/Documentos/FP/DAM/Curso%202/Materias/Programación%20de%20servicios%20y%20procesos/Evaluacion%201/Tema%202/Apuntes/PSP02/WEB/ZIP/PSP02_-4.ZIP).

Texto

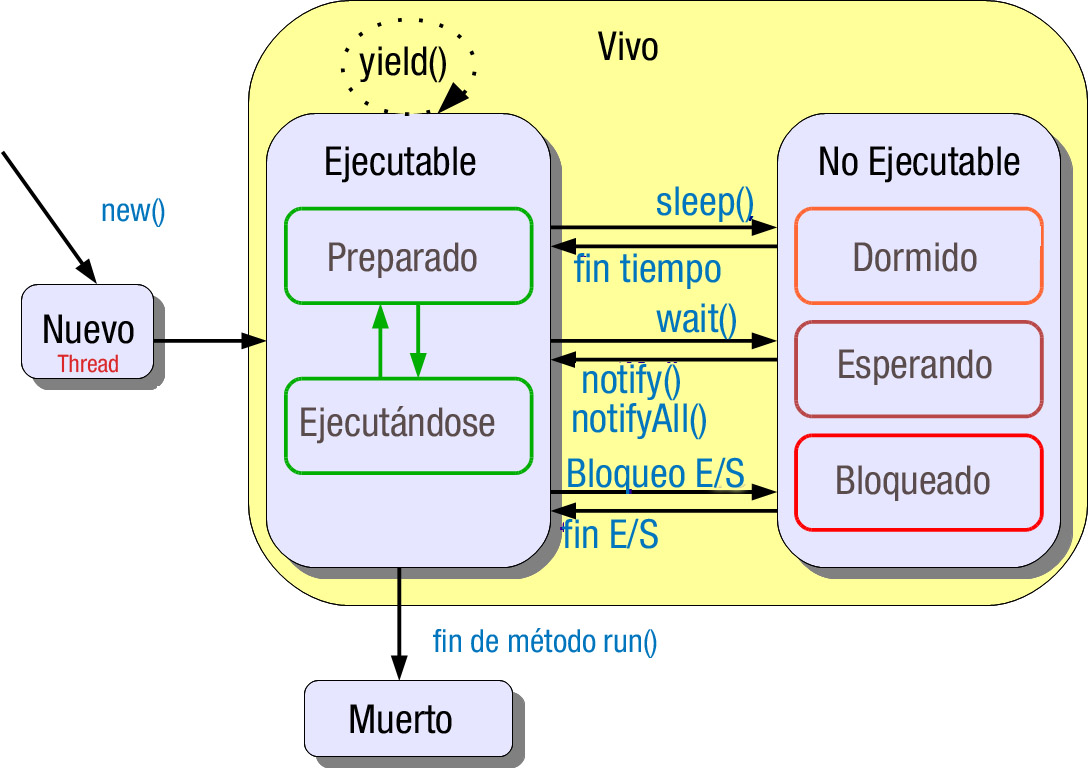
Descripción generada automáticamente

## 5.2.- Detener temporalmente un hilo.

Detener temporalmente un hilo significa que pasa al estado de “No ejecutable”.

Un hilo pasará al estado "No Ejecutable" o "Detenido" por alguna de estas circunstancias:

* **sleep()**. Se ha invocado al método **sleep**() de la clase thread, **indicando** el **tiempo** que el hilo permanecerá **dormido**. Transcurrido ese tiempo, el hilo se vuelve "Ejecutable", en concreto pasa a "Preparado".
* **El hilo está esperando**. El hilo se detiene llamando a **wait()**, y **no se reanudará**, pasará a "Ejecutable" (en concreto "Preparado") **hasta** que se llame 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; ha sido bloqueado por el sistema operativo. Cuando finaliza el bloqueo, vuelve al estado "Ejecutable", en concreto "Preparado".



El método ***suspend()*** (actualmente en desuso o deprecated) también permite detener temporalmente un hilo, y en ese caso se reanudaría mediante el método ***resume()*** (también en desuso). No uses estos métodos de la clase thread, ya que **no son seguros**. Te lo indicamos simplemente porque puede que encuentres programas que aún utilizan estos métodos.

Texto, Sitio web

Descripción generada automáticamente

## 5.3.- Finalizar un hilo.

La **forma natural de que muera** o 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:

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

**¿Hay alguna forma de comprobar si un hilo no ha muerto?**

No exactamente, pero puedes utilizar el **método *isAlive()*** de la clase ***Thread*** para comprobarlo.

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".

**Otras formas de que finalice un hilo**:

Mediante el método ***stop()***, pero es poco seguro. No debes utilizarlo. Te lo indicamos aquí simplemente porque puede que encuentres programas utilizando este método.

**Otros métodos**:

* **Thread.join**: permite decirle al programa que espere a que termine la ejecución de un hilo.

**Por ejemplo**: puedes ver un [programa](https://aula21-my.sharepoint.com/personal/5634224_alu365_murciaeduca_es/Documents/Documentos/FP/DAM/Curso%202/Materias/Programación%20de%20servicios%20y%20procesos/Evaluacion%201/Tema%202/Apuntes/PSP02/WEB/ZIP/PSP02_-5.ZIP) 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.

Texto

Descripción generada automáticamente

**Para saber más**: [razones por las que stop() es inseguro y se desaconseja](http://web.archive.org/web/20190521220606/http:/rt00149b.eresmas.net/Otras/ConcurrenciaJAVA/Deprecated.html).

Los métodos stop y stop (Throwable) (que lanzan excepciones de tipo Error, es decir, no hay que capturarlas, y por tanto el usuario no se entera de que están ahí y puede ser un poco lioso) son peligrosos. La generación de la excepción puede aparecer en cualquier momento, incluso en momentos críticos en los que el código no está preparado. Por ejemplo, es posible que salte dentro de un método sincronizado. La excepción ocasionará la finalización abrupta del método, lo que podría ocasionar que el objeto se quedara en un estado incorrecto, que podría ser visto por otras hebras, volviendo al sistema inestable.

Naturalmente una posible solución sería que en todos los métodos sincronizados o que peligran ante la aparición de la excepción la capturaran, realizaran una limpieza rápida, y luego volvieran a generar la misma excepción, para que se propague y la hebra termine.

Esta solución sin embargo es peligrosa, pues **es fácil que el programador olvide realizar el control**. Más aún, podría darse el caso de que una segunda excepción ThreadDeath saltara cuando se estaba tratando la primera, en cuyo caso el método saldría abruptamente antes de realizar la limpieza.

Una solución más aceptada es la utilización de una variable que indica a la hebra si debe detener su ejecución:

void run() {  
 while(!acabar) {  
 ...  
 }  
}  
 /\*\*  
 \* Sustituto de stop();  
 \*/ void acabar() {  
 acabar = true;  
}

## 5.4.- Ejemplo. Dormir un hilo con sleep.

**¿Por qué puede interesar dormir un hilo?**

En el siguiente ejemplo, 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.

**¿Cómo funciona el método sleep()?**

Sleep() recibe como argumento el tiempo que deseamos dormir el hilo. Cuando transcurre el tiempo especificado, el hilo vuelve a estar "Ejecutable" ("Preparado") para continuar ejecutándose.

Hay **dos formas de llamar a este método**:

* **Pasándole un argumento un entero** (positivo) que representa **milisegundos**:
  + sleep(long milisegundos)
* **Pasándole un segundo argumento** entero (entre 1 y 999999), que representa un tiempo extra en **nanosegundos** que se sumará al primer argumento.
  + sleep(long milisegundos, int nanosegundos)

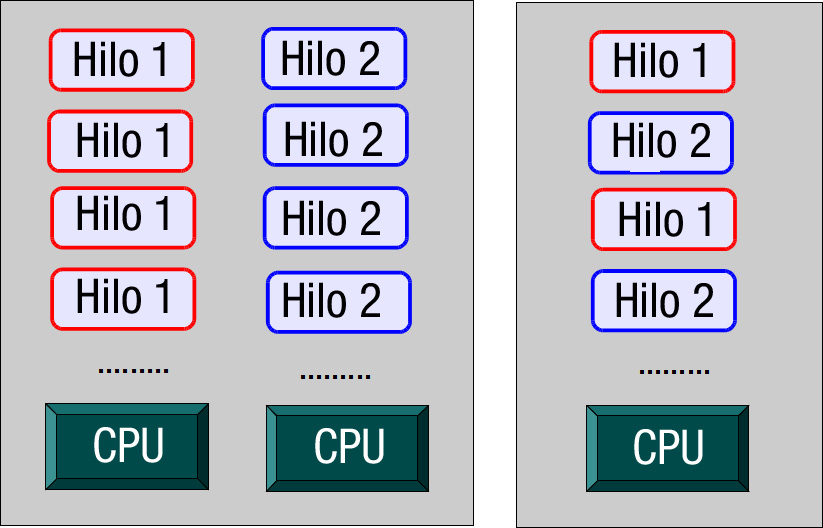
Cualquier llamada a **sleep()** **puede provocar una excepción**, que el compilador de Java nos **obliga a controlar** ineludiblemente mediante un bloque **try-catch**.

**Ejemplo**: aplicación que muestra como avanza un marcador gráfico desde 0 hasta 20. Podrás comprobar que, si no utilizamos un hilo auxiliar y lo dormimos, no podremos apreciar cómo se va incrementando el marcador.

Captura de pantalla de computadora

Descripción generada automáticamente con confianza media

# 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.
  + Requiere la planificación de una secuencia de ejecución (scheduling)

El **planificador de hilos de Java** (Scheduler), utiliza el algoritmo “fixed priority scheduling”, 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** -> algoritmo por turnos (**round-rubin**) basado en una cola circular FIFO (Primero en entrar, primero en salir).
* **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 sheudling').

Pero **la responsabilidad de ejecución de los hilos es del S.O** que corre la JVM. Y cada S.O maneja los hilos de una forma diferente:

* **S.O que implementa time-slicing** (subdivisión de tiempo), el hilo que entra en ejecución se mantiene ejecutándose un tiempo fijo o cuanto (quantum) de procesamiento. De esta forma, **su ejecución puede interrumpirse de dos formas**:
  + si otro hilo con prioridad más alta se vuelve "Ejecutable",
  + Si su "cuanto" de ejecución se acaba.

Es el patrón seguido por Linux, y por todos los Windows a partir de Windows 95 y NT.

* **S.O sin time-slicing**: el hilo que entra en ejecución, **es ejecutado hasta su muerte**; solo puede verse interrumpido de dos maneras:
  + **Si regresa a "No ejecutable"** (mediante un sleep(), wait() u operación e E/S)
  + 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.

Texto

Descripción generada automáticamente

**Para saber más**: en el siguiente enlace encontrarás información sobre [HyperThreading](https://es.wikipedia.org/wiki/Hyper-Threading) (Intel) que permite a los programas multihilo procesarlos en paralelo dentro de un único procesador.

## 6.1.- Prioridad de hilos.

La prioridad de los hilos en Java se mide en una escala entre **1 y 10**. Cuanto mayor es el valor, mayor es la prioridad.

Por defecto, el **hilo principal** (el que tiene el main()) se ejecuta 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, la que tiene por defecto el hilo donde corre el método main().

Además, en cualquier momento **se puede** **obtener y modificar la prioridad de un hilo**.

* **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 que no tienen por qué coincidir con los del sistema operativo sobre el que está corriendo. Por ello, lo mejor es que utilices en tu código sólo 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.

**Ejemplo de la gestión de prioridades de hilos**: [aquí](https://aula21-my.sharepoint.com/personal/5634224_alu365_murciaeduca_es/Documents/Documentos/FP/DAM/Curso%202/Materias/Programación%20de%20servicios%20y%20procesos/Evaluacion%201/Tema%202/Apuntes/PSP02/WEB/ZIP/PSP02_-7.ZIP).

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.

**Debes conocer**: [qué son los hilos demonio, también llamados ***servicios***](https://studylib.es/doc/592965/hilos-demonio).

* **setDaemon()**: permite establecer un hilo como hilo demonio.
* **isDaemon()**: permite saber si un hilo está establecido como hilo demonio.

Texto

Descripción generada automáticamente

**Para saber más**: [ejemplos sobre planificación y prioridad de los Threads](https://es.slideshare.net/atomeu/tema-3-63104011) (vínculo roto ☹).

## 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**: [ver el proyecto del ejemplo aquí](https://aula21-my.sharepoint.com/personal/5634224_alu365_murciaeduca_es/Documents/Documentos/FP/DAM/Curso%202/Materias/Programación%20de%20servicios%20y%20procesos/Evaluacion%201/Tema%202/Apuntes/PSP02/WEB/ZIP/PSP02_-8.ZIP).

* 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 dos 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 S.O con time-slicing, la ejecución de esos hilos se entremezclaría. Es decir, el propio S.O combatiría los hilos egoístas.

**Entonces, si el comportamiento depende del S.O, ¿qué 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**.

**Hay que 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 **S.O 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 void run(){

//se imprime 100 veces el valor de: color + i

for(int i=1;i<=100;i++)

System.out.println(color + i);

yield(); //llamada a yield()

}

}

Texto

Descripción generada automáticamente

# 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. Sin embargo, lo normal es que los distintos hilos de un programa SÍ DEBAN COMPARTIR recursos y/o información. Hay dos casos posibles:

* Dos 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.
* Dos 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.

**¿Cómo hacer que se sincronicen de forma coordinada los hilos?**

Utilizando sincronización y comunicación de hilos:

* **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 multithreadring, como son: Semaphore, CountDownLatch, CyclicBarrier y Exchanger.

En los siguientes apartados veremos ejemplos de toido 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.

[**Por ejemplo**](https://aula21-my.sharepoint.com/personal/5634224_alu365_murciaeduca_es/Documents/Documentos/FP/DAM/Curso%202/Materias/Programación%20de%20servicios%20y%20procesos/Evaluacion%201/Tema%202/Apuntes/PSP02/WEB/ZIP/PSP02_-9.ZIP): una situación que puede ocurrir cuando varios hilos actualizan una misma variable, es en el clásico “ejemplo de los jardines”. 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 una misma variable, cambiando su valor, y obteniendo de esta forma un valor no esperado de la misma.

En el proyecto de ejemplo, el recurso que comparten los diferentes hilos es la variable contador ***cuenta***. Las secciones de código donde se opera sobre esa variable son dos secciones críticas, los métodos ***incrementaCuenta()*** y ***decrementaCuenta()***.

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.

**Para saber más**: clásico ejemplo, “[El problema del barbero durmiente](https://es.wikipedia.org/wiki/Problema_del_barbero_durmiente)”, que ilustra el problema de la condición de carrera entre hilos.

## 7.2.- Monitores. Métodos syncronized.

En Java, un monitor es una porción de código protegida por un mute 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**.
* 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.

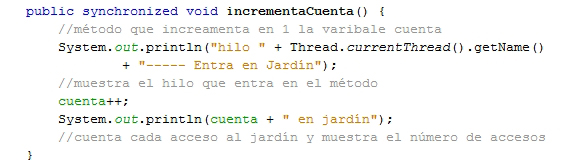
En el siguiente [documento](https://aula21-my.sharepoint.com/personal/5634224_alu365_murciaeduca_es/Documents/Documentos/FP/DAM/Curso%202/Materias/Programación%20de%20servicios%20y%20procesos/Evaluacion%201/Tema%202/Apuntes/PSP02/WEB/PDF/PSP02_CO.PDF) encontrarás una explicación detallada sobre el funcionamiento de un monitor Java.

**¿Qué bloques interesa marcar como syncronized?**

**Los que correspondan con secciones críticas** y contengan el código o datos que comparten los hilos.

En el problema anterior, se debería sincronizar tanto el método ***incrementaCuenta()***, como el ***decrementaCuenta()***, tal y como ves en el siguiente código, ya que estos métodos contienen la variable *cuenta*, la cual es modificada por diferentes hilos. Así mientras un hilo ejecuta ***jardín.incrementaCuenta()***, ningún otro hilo podrá ejecutarlo.

Aquí tienes el código corregido, junto con el [proyecto](https://aula21-my.sharepoint.com/personal/5634224_alu365_murciaeduca_es/Documents/Documentos/FP/DAM/Curso%202/Materias/Programación%20de%20servicios%20y%20procesos/Evaluacion%201/Tema%202/Apuntes/PSP02/WEB/ZIP/PSP02-10.ZIP).



[**Otro ejemplo sencillo**](https://aula21-my.sharepoint.com/personal/5634224_alu365_murciaeduca_es/Documents/Documentos/FP/DAM/Curso%202/Materias/Programación%20de%20servicios%20y%20procesos/Evaluacion%201/Tema%202/Apuntes/PSP02/WEB/ZIP/PSP02-11.ZIP) que simula el acceso simultáneo de 4 terminales a un servidor utilizando monitores Java.

**Debes conocer**: [¿Qué significa que los monitores Java son re-entrantes?](https://programacion.net/articulo/threads_de_control_100/14)

Básicamente significa que **el mismo thread puede llamar a un método sincronizado de un objeto para el que ya tiene el monitor**, es decir, puede re-adquirir el monitor.

Texto

Descripción generada automáticamente

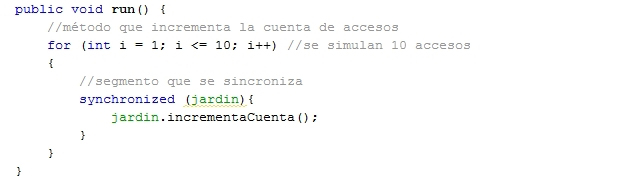
## 7.3.- Monitores. Segmentos de código. Syncronized.

Hay casos en los que no se puede, o no interesa sincronizar un método. **Por ejemplo**: no podemos sincronizar un método que no hemos creado nosotros, ya que no tenemos acceso a su código fuente.

La forma de resolver esto 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**.



**Observa que**:

* Ahora los métodos incrementaCuenta() y decrementaCuenta() no serán syncronized.
* Se está consiguiendo un acceso con exclusión mutua sobre el objeto jardin, aún cuando su clase no contiene ningún segmento ni método synchronized.

[Ejemplo del “Problema de los jardines” usando esta técnica](https://aula21-my.sharepoint.com/personal/5634224_alu365_murciaeduca_es/Documents/Documentos/FP/DAM/Curso%202/Materias/Programación%20de%20servicios%20y%20procesos/Evaluacion%201/Tema%202/Apuntes/PSP02/WEB/ZIP/PSP02-12.ZIP).

**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)**

Texto

Descripción generada automáticamente

## 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 logar 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**.

**¿Cómo funcionan realmente estos métodos?**

* **wait()** se debe invocar sobre un objeto compartido por los hilos a sincronizar.
  + Imagen que contiene Interfaz de usuario gráfica

    Descripción generada automáticamentePara poder invocar a wait() el hilo **debe tener el mutex[[1]](#footnote-1) del objeto compartido**.
  + La invocación de ***wait()*** detiene al hilo, pasa a “no ejecutable”, **lo pone en una cola de espera asociada al objeto (cola wait del objeto)** y libera el mutex del objeto.
  + El método wait() puede provocar una InterruptedException.
* **notify()**: un hilo sale de la cola de espera del objeto (cola wait), pasando al estado “ejecutable”, y se bloquea hasta conseguir el mutex del objeto para continuar su ejecución.
  + Si hay más de un hilo en al cola de espera, notify() reactivará solo a uno de ellos. El criterio de selección del hilo a reactivar (pasar a “ejecutable”) depende de la implementación de Java.
  + Una vez re-obtenido el mutex del objeto, el hilo que salió de la lista de espera continuará la ejecución del método en la instrucción siguiente a la llamada a wait().
* **notifyAll()**: Notifica a todos los hilos puestos en espera para el mismo objeto (cola wait) que ya pueden continuar.

**Sin embargo, este modelo de notificación es indirecto**: el hilo que invoca a notify() no tiene ninguna referencia del hilo que está en espera. Cuando un hilo invoca a notify(), otro hilo (no se sabe cuál de los que están en cola), es reactivado. Y con notifyAll(), se vuelven ejecutables todos los hilos que estaban bloqueados en la cola. Sin embargo, el mutex solo podrán tomarlo de uno en uno.

Imagen que contiene Texto

Descripción generada automáticamente

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

* El problema del **Productor-Consumidor**: del que ya has visto ejemplo. 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.
* El problema de los **Lectores-Escritores**: permite modelar el acceso simultáneo de varios hilos a una BB.DD, fichero u otro recurso, unos queriendo leer y otros escribir o modificar los datos.

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

**Debes conocer**: [ejemplo de un buffer circular](https://es.wikipedia.org/wiki/Buffer_circular).

Texto

Descripción generada automáticamente

## 7.5.- El problema del interbloqueo (deadlock).

Consiste en que uno a más hilos, **se bloquean o esperan indefinidamente**. La pregunta es: ¿Cómo puede llegar a ocurrir? Por ejemplo:

* 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 recurso**.

En términos simples, en las aplicaciones concurrentes **se podrá dar cuando en un hilo** haya un **bloque syncronized que llame a su vez a otro bloque syncronized**, o bien al utilizar clases de java.util.concurrent, que **llevan implícita la exclusión mutua**.

[**Ejemplo de interbloqueo**](https://aula21-my.sharepoint.com/personal/5634224_alu365_murciaeduca_es/Documents/Documentos/FP/DAM/Curso%202/Materias/Programación%20de%20servicios%20y%20procesos/Evaluacion%201/Tema%202/Apuntes/PSP02/WEB/ZIP/PSP02-14.ZIP): observa que no finaliza la ejecución del programa y que en la barra de estado de NetBeans (a la derecha) aparece la indicación de programa bloqueado. Habrá que finalizar manualmente el programa. Esto es debido a que se llama a un método syncronized desde dentro de otro.

**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.

**Debes conocer**: el interface ***Lock*** del paquete ***java.util.concurrent*** permite resolver algunos problemas de concurrencia como el interbloqueo. En el siguiente enlace encontrarás una explicación y ejemplo de uso.

En concreto, la interfaz ReadWriteLock, la cual implementa la clase ReentrantReadWriteLock.

Nos proporciona **dos tipos de locks**, **uno para lectura y otro para escritura**. Gracias a esta interfaz podremos programar de una forma muy fácil el problema de los lectores y escritores ya que **diferencia entre lector/lector, lector/escritor, y escritor/escritor** y permite mantener la lógica de estos lectores y escritores:

* Varios lectores sí es compatible.
* Un lector y un escritor no es compatible.
* Más de un escritor no es compatible.

Si no tuviéramos esta interfaz, habría que controlarlo con condicionales como hicimos cuando utilizamos monitores.

[Ver el código de un ejemplo de uso de esta técnica](https://www.redeszone.net/2012/09/10/curso-java-volumen-vii-todo-sobre-locks-en-java/).

**Para saber más**: consultar los principios de interbloqueo, ejemplos y estrategias de prevención

Texto

Descripción generada automáticamente

## 7.6.- 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:

* Llamar al constructor **Semaphore (int permisos)**:
  + Indicar el nº de permisos que se pueden dar para acceder al mismo tiempo al recurso compartido. En otras palabras: **indicar el total de hilos que podrán acceder al recurso** **simultáneamente**.
* Usar **acquire()** para solicitar acceso al recurso.
  + Podemos usar **acquire(int permisosAdquirir)** para especificar cuántos permisos se quieren consumir al mismo tiempo.
* Usar **release()** para liberar el permiso.
  + Podemos usar **release(int permisosLiberar)** para especificar cuántos permisos se quieren liberar al mismo tiempo.
* Llamar al constructor **Semaphore(int permisos, boolean justo)**:
  + El parámetro ***justo*** permite garantizar que el primer hilo en invocar ***acquire()*** 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?**

* 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.
  + [Ejemplo aquí](https://aula21-my.sharepoint.com/personal/5634224_alu365_murciaeduca_es/Documents/Documentos/FP/DAM/Curso%202/Materias/Programación%20de%20servicios%20y%20procesos/Evaluacion%201/Tema%202/Apuntes/PSP02/WEB/ZIP/PSP02-15.ZIP): Servidor web – Clientes (versión semáforos).
* Si se usa **para comunicar hilos**, en este caso un hilo invocará al método **acquire()** y otro hilo invocará al método **release()**.
  + [Ejemplo aquí](https://aula21-my.sharepoint.com/personal/5634224_alu365_murciaeduca_es/Documents/Documentos/FP/DAM/Curso%202/Materias/Programación%20de%20servicios%20y%20procesos/Evaluacion%201/Tema%202/Apuntes/PSP02/WEB/ZIP/PSP02-16.ZIP): Lector-Escritor (versión semáforos).

**Ejemplo 3**: [sincronizar hilos mediante la clase Semaphore](http://web.archive.org/web/20120228152830/https:/casidiablo.net/semaforos-java/).

Texto

Descripción generada automáticamente

**Para saber más**: [consultar otros métodos de la clase ***Semaphore***](https://docs.oracle.com/en/java/javase/13/docs/api/java.base/java/util/concurrent/Semaphore.html).

## 7.7.- La clase Exchanger.

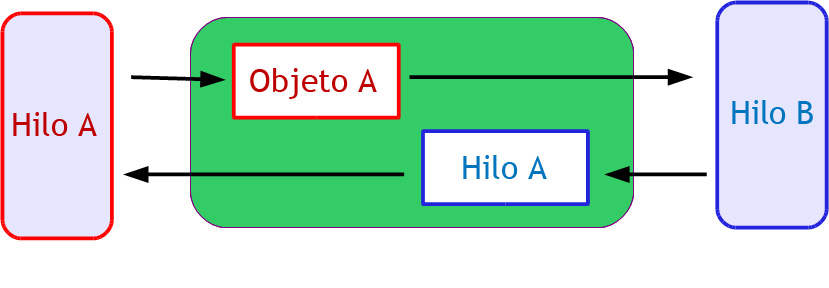
Pertenece al paquete java.util.concurrent. Establece un **punto de sincronización donde se intercambian objetos entre dos hilos**.

Existen dos **métodos** definidos en esta clase:

* **exchange (V x)**
* **exchange (V x, long timeout, TimeUnit unit)**

En ambos métodos, **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, sería el siguiente:



* Dos hilos (hiloA e hiloB) intercambiarán objetos del mismo tipo, objetoA y objetoB.
  + El hiloA invocará a exchange(**objetoA**) y el hiloB invocará a exchange(**objetoB**).
* 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)** proporciona el objeto objetoB al hiloA, y la del método **exchange(objetoB)** el objetoA al hilo B.

**¿Cuándo usar esta clase?**

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 forma que cuando el consumidor empieza a tratar la lista de datos entregados por el productor, el productor ya está produciendo una nueva lista. De esa forma, productor y consumidor trabajan concurrentemente al mismo tiempo.

[**Ejemplo**](https://aula21-my.sharepoint.com/personal/5634224_alu365_murciaeduca_es/Documents/Documentos/FP/DAM/Curso%202/Materias/Programación%20de%20servicios%20y%20procesos/Evaluacion%201/Tema%202/Apuntes/PSP02/WEB/ZIP/PSP02-17.ZIP): un hilo productor se encarga de rellenar una cadena de diez caracteres (ósea, una lista de 10 caracteres), mientras que un hilo consumidor la imprime.

Interfaz de usuario gráfica, Texto

Descripción generada automáticamente

**Para saber más**: [otro ejemplo ilustrativo de cómo utilizar la clase ***Exchanger***](http://www.java2s.com/Code/Java/Threads/AnexampleofExchanger.htm).

## 7.8.- La clase CountDownLatch.

Pertenece al paquete **java.util.concurrent**, y es una utilidad de sincronización que **permite que uno o más threads esperen hasta que otros threads finalicen su trabajo**.

**Cómo funciona**:

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

**Aspectos más importantes al usar la clase CountDownLatch**:

* **Constructor**: countDownLatch(int cuenta)
  + Parámetro “**cuenta**” indica el **total de** **hilos que deben completar su trabajo**, que será el valor de la “cuenta atrás”.
* El hilo en curso que 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.
* Si pensamos que necesitaremos reiniciar la cuenta atrás, entonces deberemos optar por usar ***CyclicBarrier***.
* El método ***getCount***() nos permitirá obtener el valor actual de la “cuenta atrás”, en caso de necesitas saberlo.

[**Ejemplo**](https://aula21-my.sharepoint.com/personal/5634224_alu365_murciaeduca_es/Documents/Documentos/FP/DAM/Curso%202/Materias/Programación%20de%20servicios%20y%20procesos/Evaluacion%201/Tema%202/Apuntes/PSP02/WEB/ZIP/PSP02-18.ZIP): de cómo usar ***CountDounLatch*** 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.

Texto

Descripción generada automáticamente

## 7.9.- La clase CyclicBarrier.

Pertenece al paquete ***java.util.concurrent***. Permite que uno o más threads se esperen hasta que todos ellos finalicen su trabajo.

**Funcionamiento**:

* **Implementa un punto de espera** que denominaremos "barrera", donde uno o más hilos esperan a que otros finalicen su trabajo.
* Finalizando 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.

**Aspectos importantes de la clase CyclicBarrier**:

* **Constructor**: CyclicBarrier(int hilosAcceden)
  + El argumento “hilosAcceden” indica el nº de hilos que van a usar la barrera. Este nº sirve para disparar la barrera.
* **Constructor**: CyclicBarrier(int hilosAcceden, Runnable accionBarrera)
  + El argumento “accionBarrera” permite que se lance una acción cuando se dispara la barrera.
* La barrera se dispara cuando llega al último hilo.
* **Cuando se dispara la barrera**, dependiendo del constructor **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**:

* El método **await(long tiempoEspera, TimeUnit unit)** funciona como en CountDownLatch, pero en este caso **el hilo espera en la barrera hasta que los demás finalicen su trabajo o se supere el** **tiempoespera**.
* **Método reset()**: permite reiniciar la barrera a su estado inicial.
* **Método getNumber Waiting()**: devuelve el nº de hilos que están en espera en la barrera.
* **Método getParties()**: devuelve el nº de hilos requieridos para esa barrera.

[**Ejemplo**](https://aula21-my.sharepoint.com/personal/5634224_alu365_murciaeduca_es/Documents/Documentos/FP/DAM/Curso%202/Materias/Programación%20de%20servicios%20y%20procesos/Evaluacion%201/Tema%202/Apuntes/PSP02/WEB/ZIP/PSP02-19.ZIP): parecido al de CountDownLatch, pero ahora resuelto con CyclicBarrier. Ahora, cada fila de la matriz 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.

Texto

Descripción generada automáticamente

**Para saber más**: [consultar otros métodos de la clase CyclicBarrier](https://docs.oracle.com/en/java/javase/13/docs/api/java.base/java/util/concurrent/CyclicBarrier.html).

# 8.- Aplicaciones multihilo.

**Caso práctico**: usar siempre las utilidades de concurrencia que proporciona Java repercutirá en la eficiencia y facilidad de mantenimiento de la aplicación.

**Propiedades de una aplicación multihilo**:

* **Seguridad**: la app no debe llegar 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 por culpa de un hilo. Esto implica un comportamiento no egoísta de los hilos y ausencia de interbloqueos e inanición.

**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.

**Aspectos a tener en cuenta al desarrollar aplicaciones multihilo**:

* **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 -> usar yield() siempre que no se use wait() y notify().
* 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**: nos proporciona clases.
* **Mayor rendimiento**: algoritmos desarrollados por expertos en concurrencia y rendimiento.
* **Mayor fiabilidad**: al 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** concurrente.
* **Menor mantenimiento**. El código que generemos será más legible y fácil de actualizar.
* **Mayor productividad**: al usar una API, nos coordinamos mejor 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**.

**Colecciones importantes**:

* **La interfaz Queue**. Es una colección diseñada para **almacenar elementos antes de procesarlos**, ofreciendo diferentes operaciones como inserción, extracción e inspección.
* **La interfaz BlockingQueue**, 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** **Syncronized**, pero proporciona **métodos** que hacen **más fácil el uso de bloqueos y condiciones**. Entre ellos.

* El método **newCondition()**: permite tener un **mayor control sobre el bloqueo** y genera un objeto del tipo **Condition**.
  + 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 ***java.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.

**Para saber más**: Tutorial sobre hilos de Oracle, incluyendo ejemplos de uso de las utilidades incluidas en el paquete **java.util.concurrent** para aplicaciones multihilo [High Level Concurrency Objects (The Java™ Tutorials > Essential Java Classes > Concurrency) (oracle.com)](https://docs.oracle.com/javase/tutorial/essential/concurrency/highlevel.html), [Threads « Java (java2s.com)](http://www.java2s.com/Code/Java/Threads/CatalogThreads.htm)

## 8.2.- La interfaz Executor y los pools de hilos.

Las aplicaciones tipo **servidor** tienen que **atender un nº masivo y concurrente de peticiones** de usuario, en forma de tareas que deben ser procesadas lo antes posible.

Sin embargo, **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)**, funciona como un **contenedor** dentro del cual se crean y se inician un nº 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 nº 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.
* **newSheduledExecutor()**: crea un pool que va a ejecutar **tareas programadas cada cierto tiempo**, ya sea una sola vez o de manera repetida.
  + Es **parecido a un Timer**, pero con varios threads en lugar de uno solo, 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).

También está el método submit, que parece que hace lo mismo que execute, pero puedes obtener respuesta de la ejecución del hilo, en caso de que el hilo devuelva algo.

La interfaz **ExecutorService** tiene una serie de **métodos para controlar 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.

**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.

**Para saber más**: otro ejemplo de **Executor**, así como de otras clases del paquete **java.util.concurrent**: [Java ExecutorService (jenkov.com)](https://jenkov.com/tutorials/java-util-concurrent/executorservice.html)

Interfaz de usuario gráfica, Texto

Descripción generada automáticamente

## 8.3.- Gestión de excepciones.

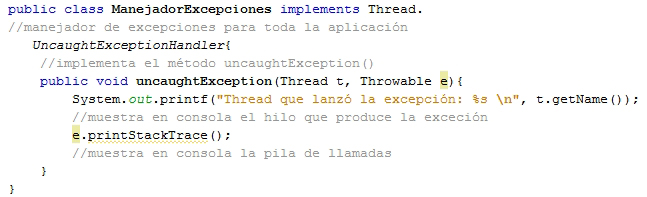
Para gestionar las excepciones de una aplicación multihilo puedes utilizar el método **uncaughtExceptionHandler()** de la clase thread, que **permite definir un manejador de excepciones**.

**Cómo crear un manejador de excepciones**:

* Creamos 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:

* El manejador solo mostrará qué hilo ha producido la excepción y la pila de llamadas de la excepción.



En el siguiente enlace está el proyecto completo. Se creará el anterior manejador de excepciones y se implementará un hilo que divide el número 100 por un número aleatorio comprendido entre 0 y 4, dando así la posibilidad de dividir por 0. Se crean e inician 5 hilos que harán uso del manejador.

Texto

Descripción generada automáticamente

## 8.4.- Depuración y documentación.

### 8.4.1.- Depuración.

Dos cosas fundamentales para desarrollar 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.

**Ejemplo**: [cómo depurar aplicaciones multihilo desde NetBeans](https://netbeans.apache.org/tutorial/main/kb/docs/java/debug-multithreaded/).

### 8.4.2.- Documentación.

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

**Utilizaremos**: JavaDoc: para que de forma automática, genere la documentación de la aplicación a partir del código fuente.

* [Generating JavaDoc in Netbeans IDE 6.5 - YouTube](https://www.youtube.com/watch?v=UK_7XQZhhoA)

1. Mutex: mecanismo de sincronización que permite a un solo subproceso o proceso acceder a un recurso compartido, como una memoria o un archivo, evitando así conflictos o inconsistencias. Para entenderlo, la metáfora sería algo como mutex ≈ llave del monitor, es decir, la llave que permite bloquear y desbloquear un recurso. [↑](#footnote-ref-1)