

# Programación de Servicios y Procesos

Tema 2 Programación Multihilo

José Luis González Sánchez





# Contenidos

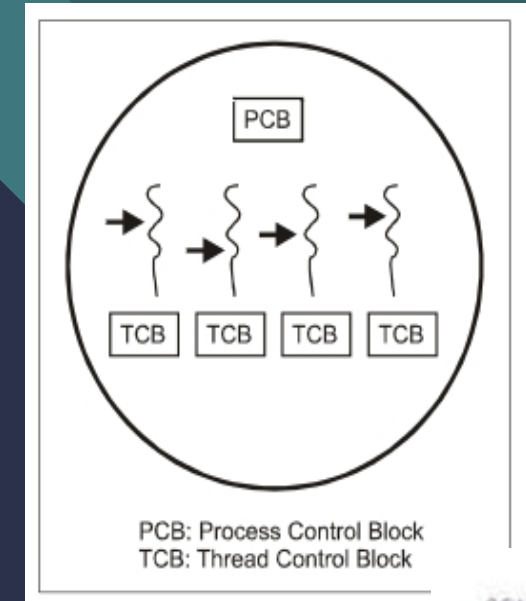
1. Programación Multihilo
2. Estados de un Hilo
3. Programación de Hilos
4. Mecanismos de Sincronización
5. Ejemplos

# Programación Multihilo

Poder hacer varias cosas a la vez

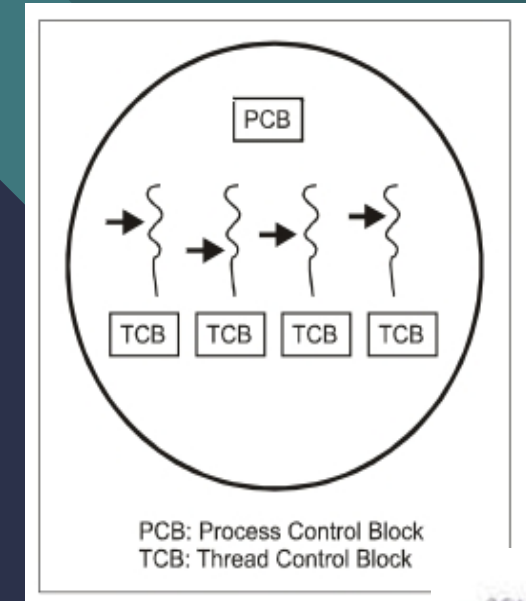
# Programación Multihilo

- Programa de flujo único. Es aquel que realiza las actividades o tareas que lleva a cabo una a continuación de la otra, de manera secuencial, lo que significa que cada una de ellas debe concluir por completo, antes de que pueda iniciarse la siguiente.
- Programa de flujo múltiple. Es aquel que coloca las actividades a realizar en diferentes flujos de ejecución, de manera que cada uno de ellos se inicia y termina por separado, pudiéndose ejecutar éstos de manera simultánea o concurrente.
- La programación multihilo o multithreading consiste en desarrollar programas o aplicaciones de flujo múltiple. Cada uno de esos flujos de ejecución es un thread o hilo.
- Desde este punto de vista, un proceso no se ejecuta, sino que solo es el espacio de direcciones donde reside el código que es ejecutado mediante uno o más hilos.
- Por lo tanto podemos hacer las siguientes observaciones:
  - Un hilo no puede existir independientemente de un proceso.
  - Un hilo no puede ejecutarse por si solo.
- Dentro de cada proceso puede haber varios hilos ejecutándose. Un único hilo es similar a un programa secuencial; por si mismo no nos ofrece nada nuevo. Es la habilidad de ejecutar varios hilos dentro de un proceso lo que ofrece algo nuevo y útil; ya que cada uno de estos hilos puede ejecutar actividades diferentes al mismo tiempo. Así en un programa un hilo puede encargarse de la comunicación con el usuario, mientras que otro hilo transmite un fichero, otro puede acceder a recursos del sistema (cargar sonidos, leer ficheros, ...), etc.



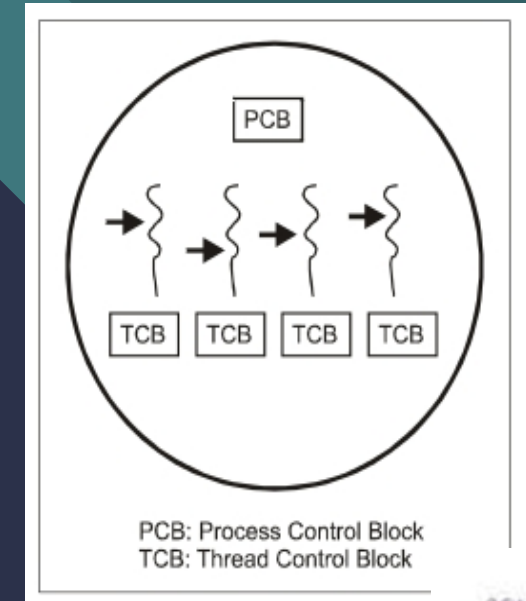
# Programación Multihilo. Concurrency

- Desde el punto de vista de las aplicaciones los threads son útiles porque permiten que el flujo del programa sea dividido en dos o más partes, cada una ocupándose de alguna tarea. Por ejemplo un Thread puede encargarse de la comunicación con el usuario, mientras otros actúan en segundo plano, realizando la transmisión de un fichero, accediendo a recursos del sistema (cargar sonidos, leer ficheros ...), etc.
- De hecho todos los programas con interface gráfico son multithread porque los eventos y las rutinas de dibujo de las ventanas corren en un thread distinto al principal.
- **Resumiendo, cada hilo se ejecuta en forma estrictamente secuencial y tiene su propia pila, el estado de los registros de la CPU y su propio contador de programa. En cambio, comparten el mismo espacio de direcciones, lo que significa compartir también las mismas variables globales, el mismo conjunto de ficheros abiertos, procesos hijos (no hilos hijo), señales, semáforos y cualquier mecanismo de sincronización.**
- **Un thread no puede existir independientemente de un programa, sino que se ejecuta dentro de un programa o proceso.**



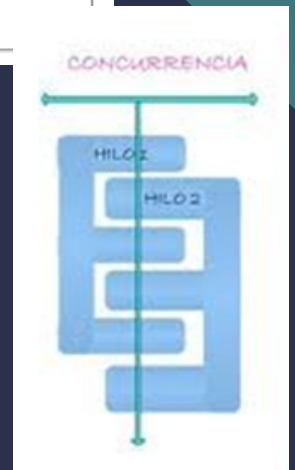
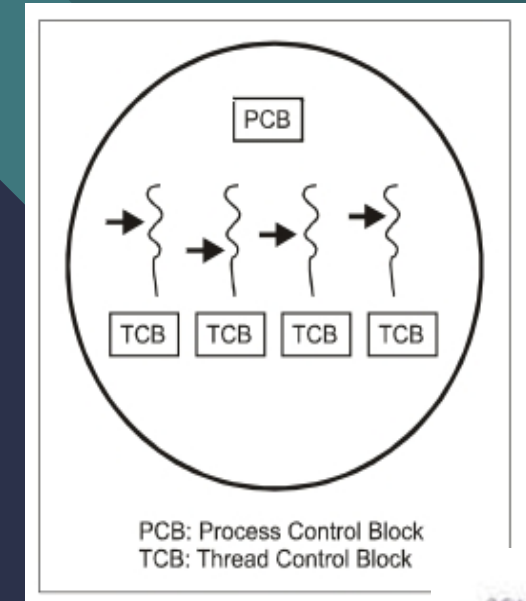
# Programación Multihilo. Compartición de Recursos

- Todos los hilos de proceso que pertenecen a un mismo proceso comparten un área común de datos que sirve para intercambiar información entre ellos.
- No es necesario acudir a técnicas de comunicación entre procesos tales como paso de mensajes, ya que todos los hilos son capaces de acceder directamente a los datos compartidos.
- Por otro lado, la conmutación entre hilos de un mismo proceso es muy rápida, puesto que la cantidad de información que ha de ser salvada y/o restaurada por el sistema es mucho menor. Por eso, cuando se trata con hilos siempre se habla de cambios de contexto ligeros, en contraposición a los cambios de contexto pesados, que implican el manejo de procesos.
- 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.
- Seguro que te estarás preguntando "si los hilos de un proceso comparten el mismo espacio de memoria, ¿qué pasa si uno de ellos la corrompe?" La respuesta es, que los otros hilos también sufrirán las consecuencias.



# Programación Multihilo. Ventajas

- Como consecuencia de compartir el espacio de memoria, los hilos aportan las siguientes ventajas sobre los procesos:
  - Se consumen menos recursos en el lanzamiento, y la ejecución de un hilo que en 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.

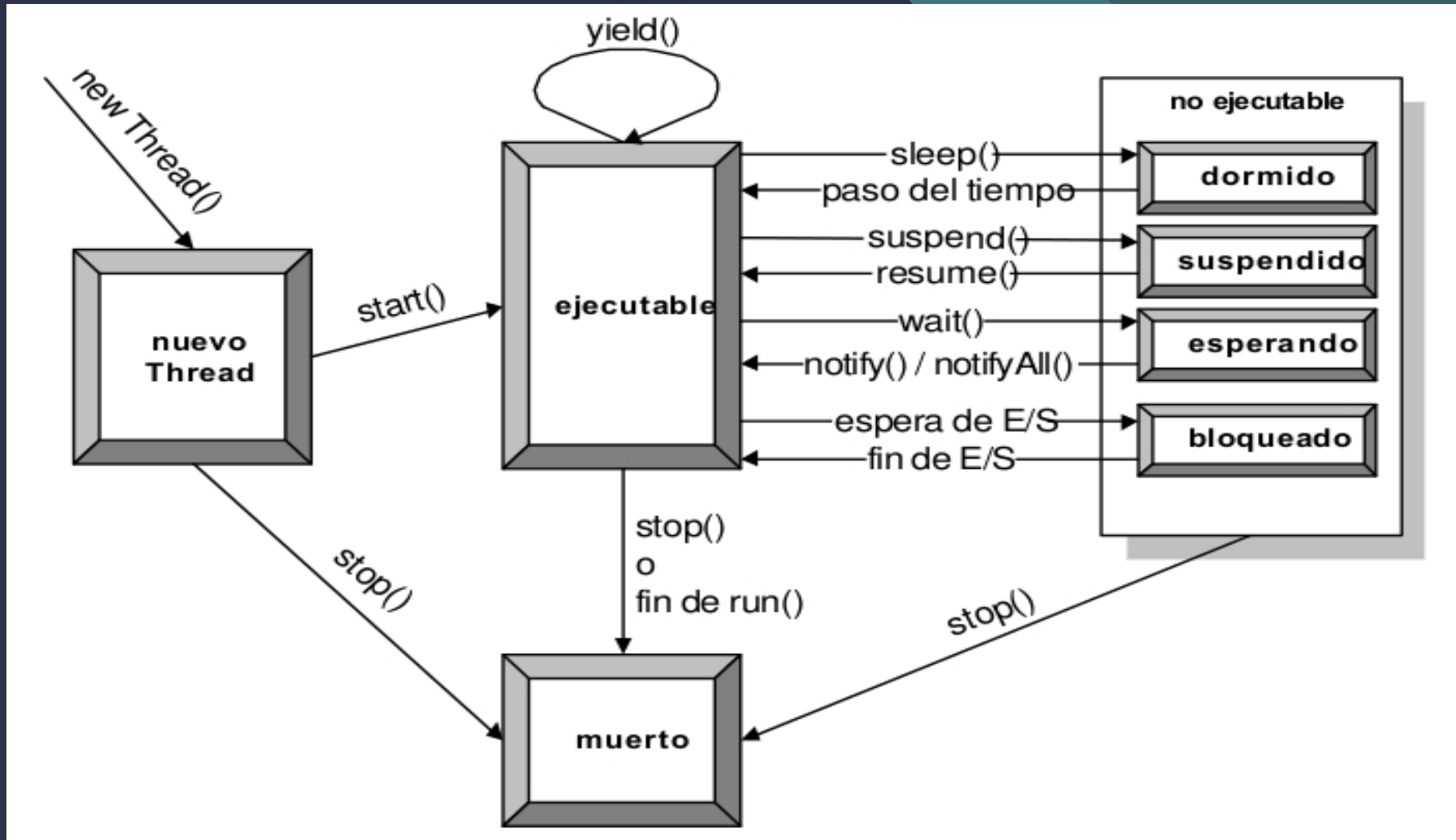


# Estados de un hilo

Todo tiene una vida



# Estados de un hilo

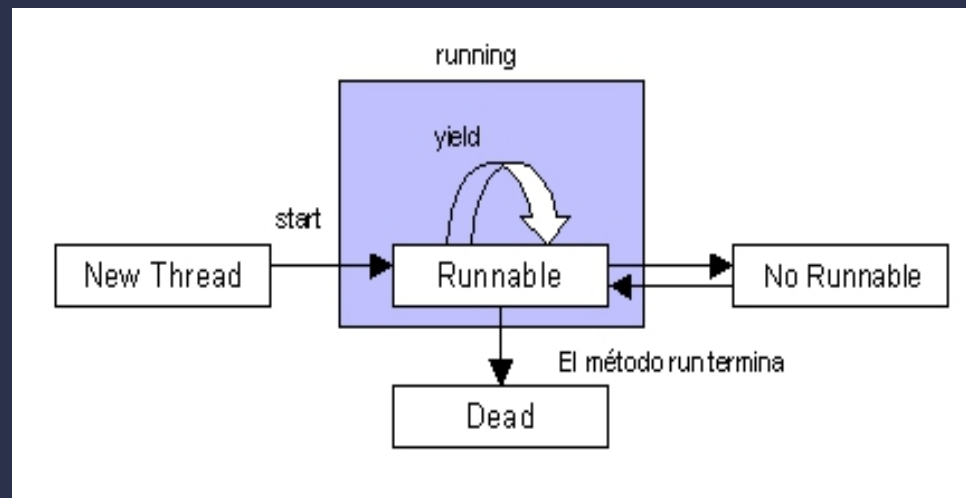


# Estados de un hilo

- **New:** Un hilo esta en el estado new la primera vez que se crea y hasta que el método start es llamado. Los hilos en estado new ya han sido inicializados y están listos para empezar a trabajar, pero aún no han sido notificados para que empiecen a realizar su trabajo.
- **Runnable:** Cuando se llama al método start de un hilo nuevo, el método run es invocado y el hilo entra en el estado runnable. Este estado podría llamarse “running” porque la ejecución del método run significa que el hilo está corriendo. Sin embargo, debemos tener en cuenta la prioridad de los hilos. Aunque cada hilo está corriendo desde el punto de vista del usuario, en realidad todos los hilos, excepto el que en estos momentos está utilizando la CPU, están en el estado runnable (ejecutables, listos para correr) en cualquier momento dado. Uno puede pensar conceptualmente en el estado runnable como si fuera el estado “running”, sólo tenemos que recordar que todos los hilos tienen que compartir los recursos del sistema.
- **Not running:** El estado not running se aplica a todos los hilos que están parados por alguna razón. Cuando un hilo está en este estado, está listo para ser usado y es capaz de volver al estado runnable en un momento dado. Los hilos pueden pasar al estado not running a través de varias vías. A continuación se citan diferentes eventos que pueden hacer que un hilo esté parado de modo temporal.
  - El método suspend ha sido llamado
  - El método sleep ha sido llamado
  - El método wait ha sido llamado
  - El hilo está bloqueado por I/O
- Para cada una de estas acciones que implica que el hilo pase al estado not running hay una forma para hacer que el hilo vuelva a correr. A continuación presentamos la lista de eventos correspondientes que pueden hacer que el hilo pase al estado runnable.
  - Si un hilo está suspendido, la invocación del método resume
  - Si un hilo está durmiendo, pasarán el número de milisegundos que se ha especificado que debe dormir
  - Si un hilo está esperando, la llamada a notify o notifyAll por parte del objeto por el que espera
  - Si un hilo está bloqueado por I/O, la finalización de la operación I/O en cuestión

# Estados de un hilo

- **Dead** : Un hilo entra en estado dead cuando ya no es un objeto necesario. Los hilos en estado dead no pueden ser resucitados y ejecutados de nuevo. Un hilo puede entrar en estado dead a través de dos vías:
  - El método run termina su ejecución.
  - El método stop es llamado.
- La primera opción es el modo natural de que un hilo muera. Uno puede pensar en la muerte de un hilo cuando su método run termina la ejecución como una muerte por causas naturales.
- En contraste a esto, está la muerte de un hilo “por causa” de su método stop. Una llamada al método stop mata al hilo de modo asíncrono.



# Programación de Hilos

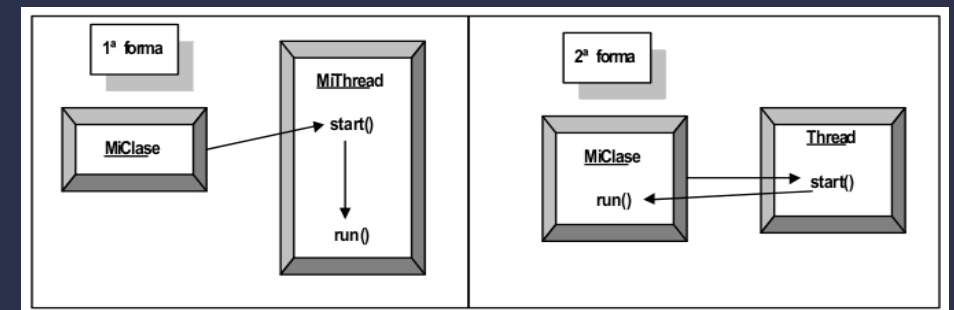
Concurrencia y Asincronía

# Programando Hilos

- El lenguaje de programación Java proporciona soporte para hilos a través de una simple interfaz y un conjunto de clases. La interfaz de Java y las clases que incluyen funcionalidades sobre hilos son las siguientes:
  - Thread
  - Runnable
  - ThreadDeath
  - ThreadGroup
  - Object
  - Executor y ExecutorService

# Programando Hilos

- La clase **Thread** es la clase responsable de producir hilos funcionales para otras clases. Para añadir la funcionalidad de hilo a una clase simplemente se deriva la clase de Thread y se ignora el método run. Es en este método run donde el procesamiento de un hilo toma lugar, y a menudo se refieren a él como el cuerpo del hilo. La clase Thread también define los métodos start y stop, los cuales te permiten comenzar y parar la ejecución del hilo, además de un gran número de métodos útiles.
  - <https://docs.oracle.com/javase/7/docs/api/java/lang/Thread.html>
- La clase **Runnable**. Java no soporta herencia múltiple de forma directa, es decir, no se puede derivar una clase de varias clases padre. Esto nos plantea la duda sobre cómo podemos añadir la funcionalidad de Hilo a una clase que deriva de otra clase, siendo ésta distinta de Thread. Para lograr esto se utiliza la interfaz Runnable. La interfaz Runnable proporciona la capacidad de añadir la funcionalidad de un hilo a una clase simplemente implementando la interfaz, en lugar de derivándola de la clase Thread. Las clases que implementan la interfaz Runnable proporcionan un método run que es ejecutado por un objeto hilo asociado que es creado aparte. Esta es una herramienta muy útil y a menudo es la única salida que tenemos para incorporar multihilo dentro de las clases. Esta cuestión será tratada más ampliamente en el apartado de creación de hilos.
  - <https://docs.oracle.com/javase/7/docs/api/java/lang/Runnable.html>

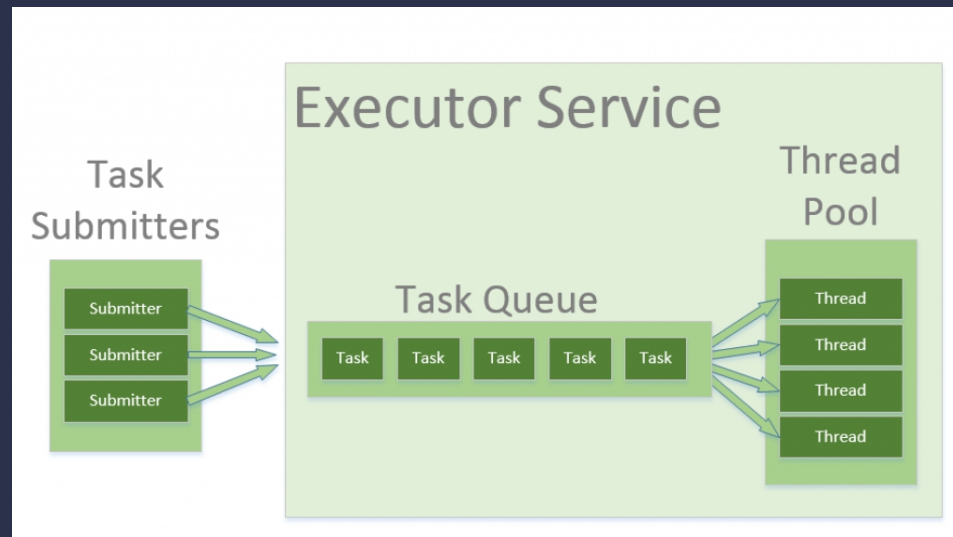


# Programando Hilos

- La clase de error **ThreadDeath** proporciona un mecanismo que permite hacer limpieza después de que un hilo haya sido finalizado de forma asíncrona. Se llama a ThreadDeath una clase error porque deriva de la clase Error, la cual proporciona medios para manejar y notificar errores. Cuando el método stop de un hilo es invocado, una instancia de ThreadDeath es lanzada por el moribundo hilo como un error. Sólo se debe recoger el objeto ThreadDeath si se necesita para realiza una limpieza específica a la terminación asíncrona, lo cual es una situación bastante inusual. Si se recoge el objeto, debe ser relanzado para que el hilo realmente muera.
  - <https://docs.oracle.com/javase/8/docs/api/java/lang/ThreadDeath.html>
- La clase **ThreadGroup** es la implementación del concepto de grupo de hilos en Java. Ofrece, por tanto, la funcionalidad necesaria para la manipulación de grupos de hilos para las aplicaciones Java. Un objeto ThreadGroup puede contener cualquier número de hilos. Los hilos de un mismo grupo generalmente se relacionan de algún modo, ya sea por quién los creó, por la función que llevan a cabo, o por el momento en que deberían arrancarse y parar. El grupo de hilos de más alto nivel en una aplicación Java es el grupo de hilos denominado main.
  - <https://docs.oracle.com/javase/7/docs/api/java/lang/ThreadGroup.html>
- Aunque, estrictamente hablando, no es una clase de apoyo a los hilos, la clase **Object** proporciona unos cuantos métodos cruciales dentro de la arquitectura multihilo de Java. Estos métodos son **wait**, **notify** y **notifyAll**.
  - El método wait hace que el hilo de ejecución espere en estado dormido hasta que se le notifique que continúe.
  - Del mismo modo, el método notify informa a un hilo en espera de que continúe con su ejecución.
  - El método notifyAll es similar a notify excepto que se aplica a todos los hilos en espera. Estos tres métodos solo pueden ser llamados desde un método o bloque sincronizado (o bloque de sincronización).

# Programando Hilos. Patrón Thread Pool

- En Java, los hilos se asignan a hilos a nivel del sistema, que son recursos del propio sistema operativo. Si creamos hilos de manera incontrolable, es posible que nos quedemos sin estos recursos rápidamente.
- El sistema operativo también hace el cambio de contexto entre hilos y subprocesos, para emular el paralelismo. Una visión simplista es que cuantos más hilos/subprocesos generamos, menos tiempo pasa cada subproceso/hilo haciendo el trabajo real.
- El patrón Thread Pool ayuda a ahorrar recursos en una aplicación multiproceso/multihilo y a contener el paralelismo en ciertos límites predefinidos.
- Cuando usamos un grupo de subprocesos o hilos, escribimos nuestro código concurrente en forma de tareas paralelas y las enviamos para su ejecución a una instancia de un grupo de subprocesos/hilos. Esta instancia controla varios subprocesos/hilos reutilizados para ejecutar estas tareas y los va procesando a partir de una cola.



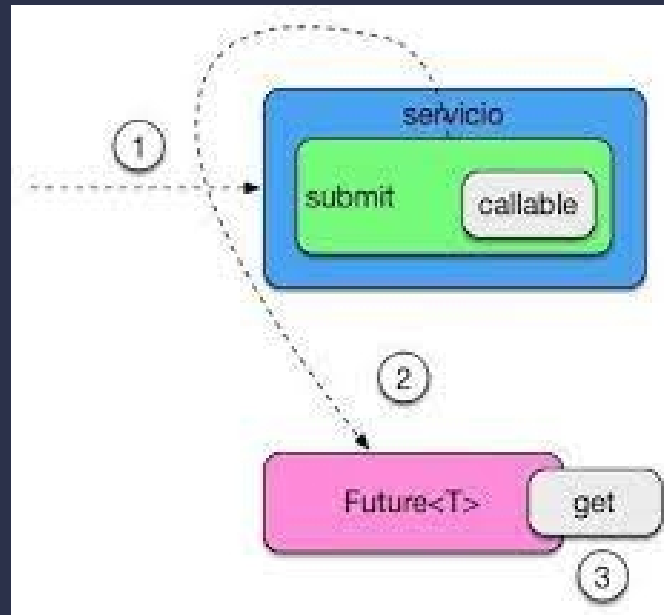


# Programando Hilos. Concurrencia. Patrón Thread Pool

- La interfaz del **Executor** tiene un método de ejecución único para enviar instancias ejecutables para su ejecución.
- La interfaz `ExecutorService` tiene distintos métodos para controlar el progreso de las tareas y manejar la terminación del servicio.
- ¿Por qué? Porque crear hilos es costoso y si estamos creando los mismos hilos que hacen lo mismo estamos malgastando recursos. Debemos poder agruparlos, cachearlos, reutilizarlos, etc,
- El Pool de Hilos permite crear un conjunto de hilos que se van procesando dentro de una cola conforme se van completando los anteriores, de esta forma por ejemplo puedo asignar 100 trabajos a una cola de 10 de modo que cada vez que se libera uno va procesando el siguiente.

# Programando Hilos. Asincronía. Callable y Future

- Anteriormente hemos visto como usar la interfaz Runnable para ejecutar tareas dentro de los hilos. Pero está limitado porque no podemos devolver un valor.
- **Callable es parecido a Runnable pero con la ventaja de que devuelve un valor.** Además podemos usar expresiones Lambda.
- Para ejecutar Callables dentro de un Pool o ExecutorService debemos obtener el resultado usando Future.
- **Future es una promesa**, es decir, es el resultado de una operación asíncrona. Representa el resultado de una tarea que se completará en el futuro. Es entonces, una vez terminada cuando debemos procesar lo que nos devuelve. Para ello trabajaremos con los métodos get e isDone().

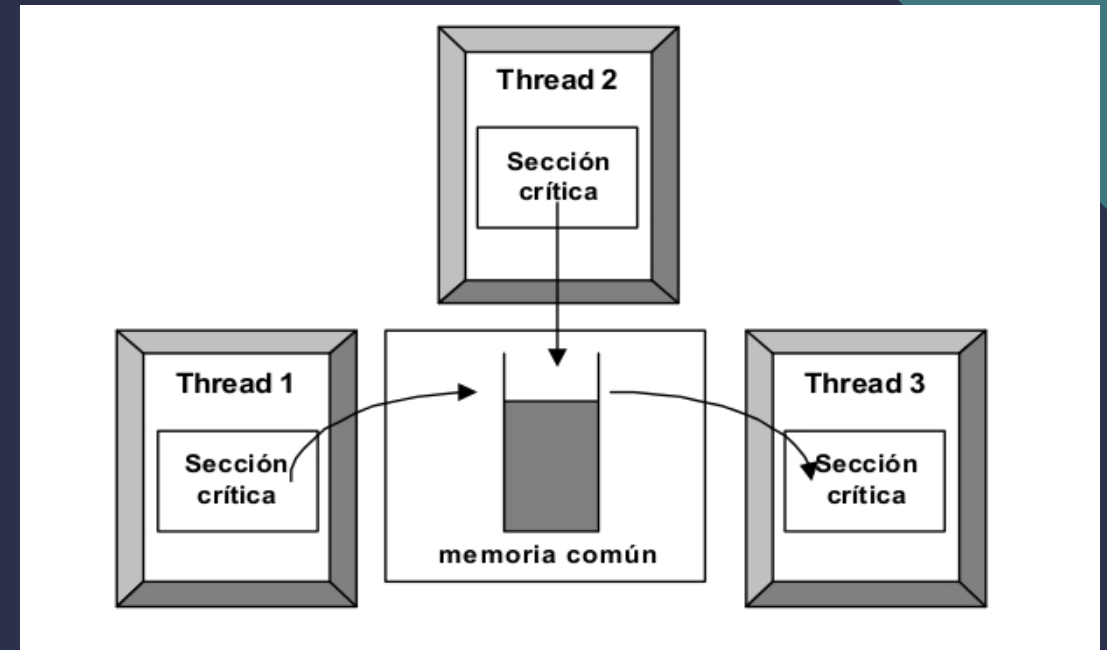


# Mecanismos de Sincronización

Manejando la concurrencia

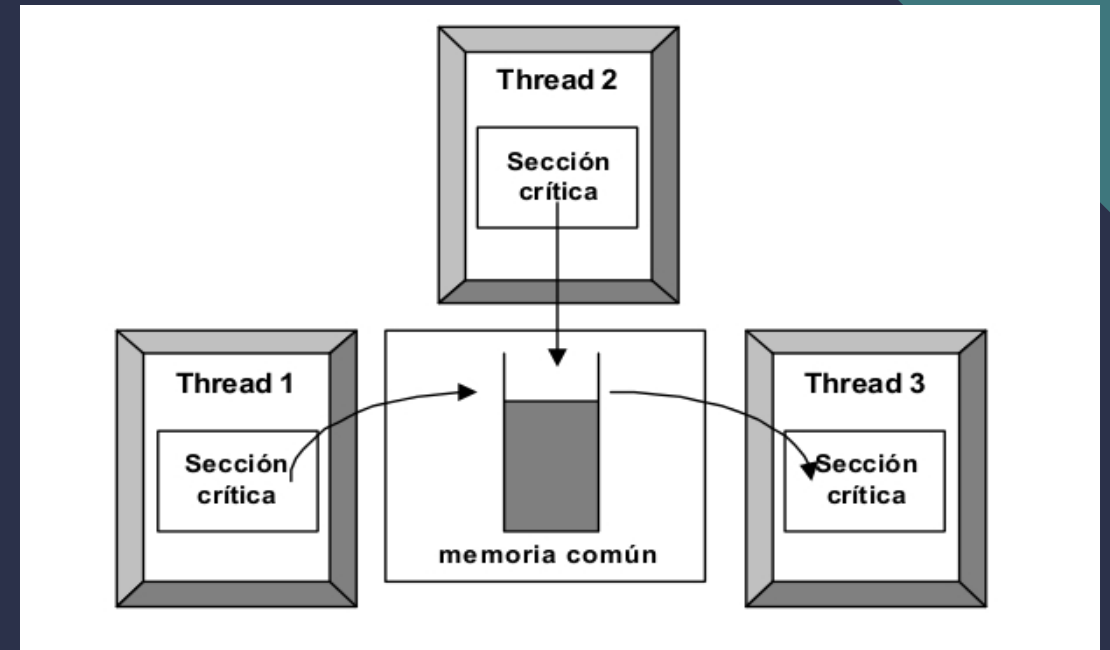
# Mecanismos de Sincronización. Metodo Synchronized

- Hasta el momento se ha estado tratando con threads asíncronos, cada uno de ellos se ejecutaba independientemente de los demás, que se ejecutaban concurrentemente. Existen, sin embargo, situaciones en las que es imprescindible sincronizar en cierta forma la ejecución de distintos threads que buscan un objetivo común, cooperando entre ellos para su correcto funcionamiento.
- Para ello debemos tener en cuenta los recursos compartidos, y el problema de la sección crítica
- Se llama sección crítica a los segmentos de código, dentro de un programa, que acceden a zonas de memoria comunes desde distintos threads que se ejecutan concurrentemente.
- En Java, las secciones críticas se marcan con la palabra reservada **synchronized**. Aunque está permitido marcar bloques de código más pequeños que un método como synchronized, para seguir una buena metodología de programación, es preferible hacerlo a nivel de método.



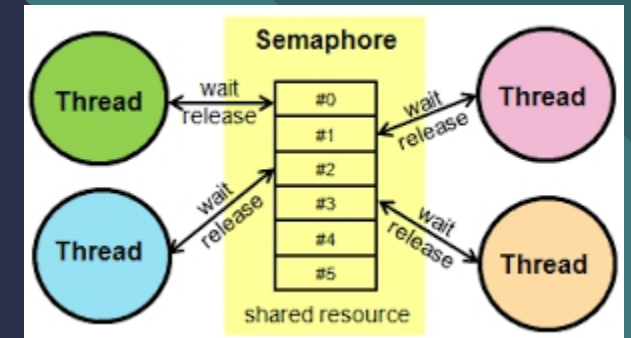
# Mecanismos de Sincronización. Cerrojos y Variables atómicas

- Cerrojos, simplemente cierran el paso al recurso compartido hasta que terminan y lo vuelven a abrir. Pueden ser reentrantes o de acceso de escritura lectura (Problema de lectores escritores)
- Las variables atómicas son aquellas que su acceso y modificación se hacen de manera atómica mediante métodos get/set
  - AtomicInteger
  - AtomicLong
  - AtomicBoolean
  - LongAdder
  - DoubleAdder



# Mecanismos de Sincronización. Semáforos

- Un semáforo binario es un indicador (S) de condición que registra si un recurso está disponible o no. Un semáforo binario sólo puede tomar dos valores: 0 y 1. Si, para un semáforo binario,  $S = 1$  entonces el recurso está disponible y la tarea lo puede utilizar; si  $S = 0$  el recurso no está disponible y el proceso debe esperar.
- Los semáforos se implementan con una cola de tareas o de condición a la cual se añaden los procesos que están en espera del recurso. Sólo se permiten tres operaciones sobre un semáforo:
  - Inicializar
  - Espera (wait)
  - Señal (signal)
- En JAVA usamos la clase Semaphore, del paquete java.util.concurrent.
- os serán:
  - Lo primero es el método acquire() de la clase Semaphore. Este método bloquea el semáforo (wait).
  - El método release() de la clase Semaphore, libera el semáforo para los demás procesos (signal).



# Mecanismos de Sincronización. Monitores

- Un monitor, es una estructura de datos que monitoriza la sección crítica. Es el encargado de hacer esperar, notificar a uno o a todos si está libre y asegurarse mediante espera activa que todos los interesados acceden al recurso de manera equitativa y segura, sin producir innanición y de forma exclusiva.
- En Java usaremos la notación synchronized para crear los métodos del monitor y además:
  - wait(long millis): espera hasta una notificación, o hasta que pasen <millis> milisegundos. Si el parámetro es 0, la espera es infinita.
  - wait(long tiempo, int nanos): igual que la anterior, pero con precisión de nanosegundos.
  - wait(): igual que wait(0). Es la explicada anteriormente.
  - notify(): despierta a una hebra de las que están esperando.
  - notifyAll(): despierta a todas las hebras que están esperando.
- Colecciones concurrentes. Nos evitan reinventar la rueda, programando soluciones como la del productor–consumidor. Existen diferentes tipos, entre los que están:
  - BlockingQueue: estructura FIFO que bloquea si la cola se queda llena o vacía.
  - ConcurrentMap: Map con operaciones atómicas.
  - ConcurrentNavigableMap: NavigleMap con búsquedas aproximadas.

## wait/notify


```
// Thread 1
synchronized(lock) {
    while(! someCondition()) {
        lock.wait();
    }
}

// Thread 2
synchronized(lock) {
    satisfyCondition();
    lock.notifyAll();
}
```

**You must synchronize.**

**Always wait in a loop.**

**Synchronize here too!**



# Ejemplos

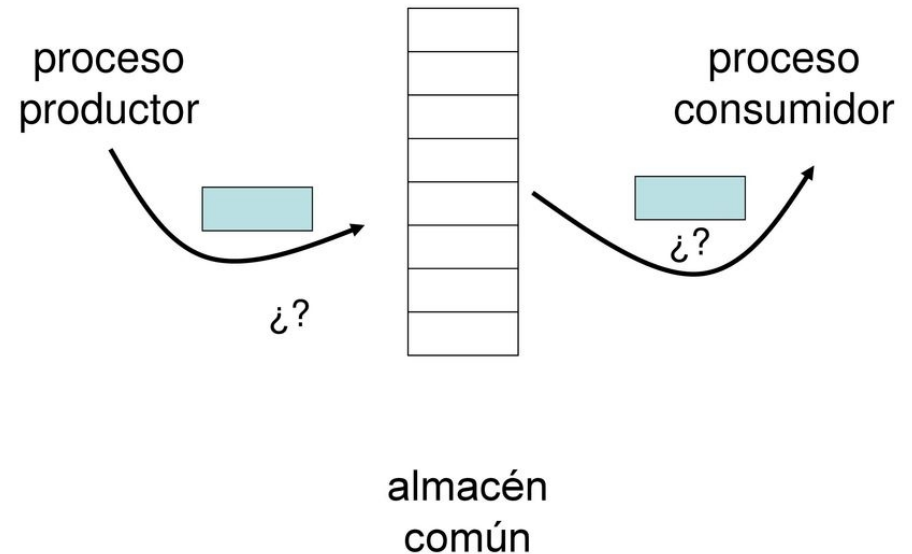
En esta vida todo se repite o sigue un patrón



# Ejemplos. Productor - Consumidor

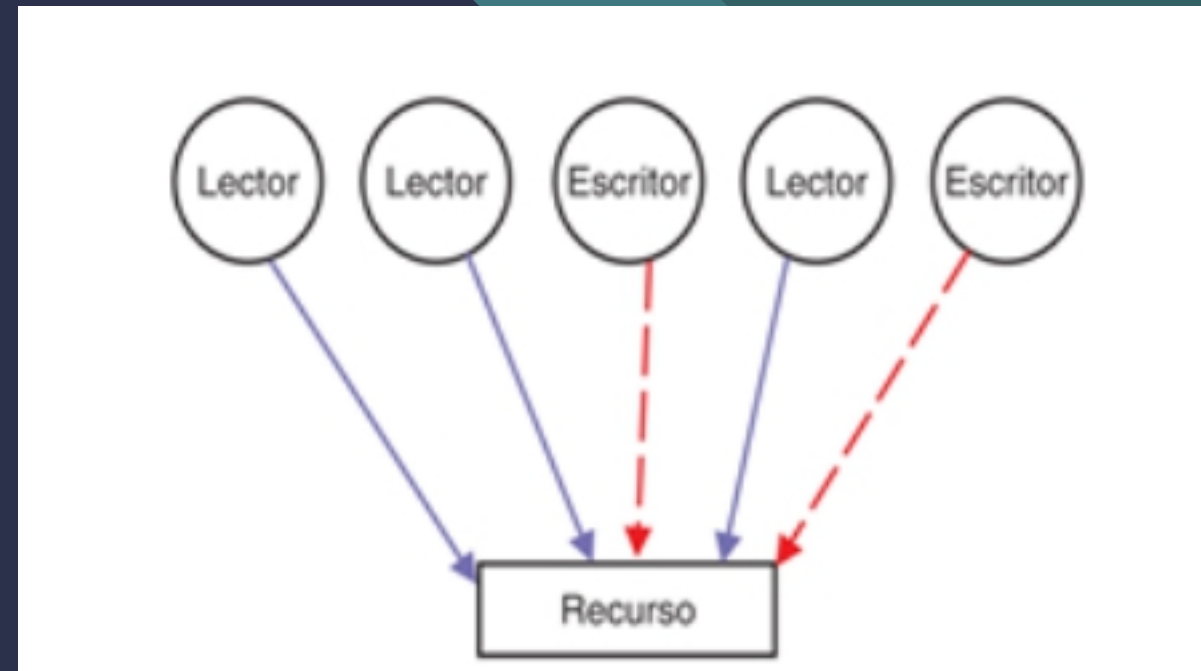
- El problema del **productor-consumidor** es un ejemplo clásico de problema de sincronización de multiprocesos.
- El programa describe dos procesos, **productor y consumidor**, ambos comparten un **buffer de tamaño finito**. La tarea del **productor es generar un producto, almacenarlo y comenzar nuevamente**; mientras que el **consumidor toma (simultáneamente) productos uno a uno**. El problema consiste en que **el productor no añada más productos que la capacidad del buffer y que el consumidor no intente tomar un producto si el buffer está vacío**.
- La idea para la solución es la siguiente, ambos procesos (productor y consumidor) se ejecutan simultáneamente y se “despiertan” o “duermen” según el estado del buffer.
- Concretamente, el productor agrega productos mientras quede espacio y en el momento en que se llene el buffer se pone a “dormir”. Cuando el consumidor toma un producto notifica al productor que puede comenzar a trabajar nuevamente. En caso contrario, si el buffer se vacía, el consumidor se pone a dormir y en el momento en que el productor agrega un producto crea una señal para despertarlo.

## Ejemplo: problema del productor-consumidor



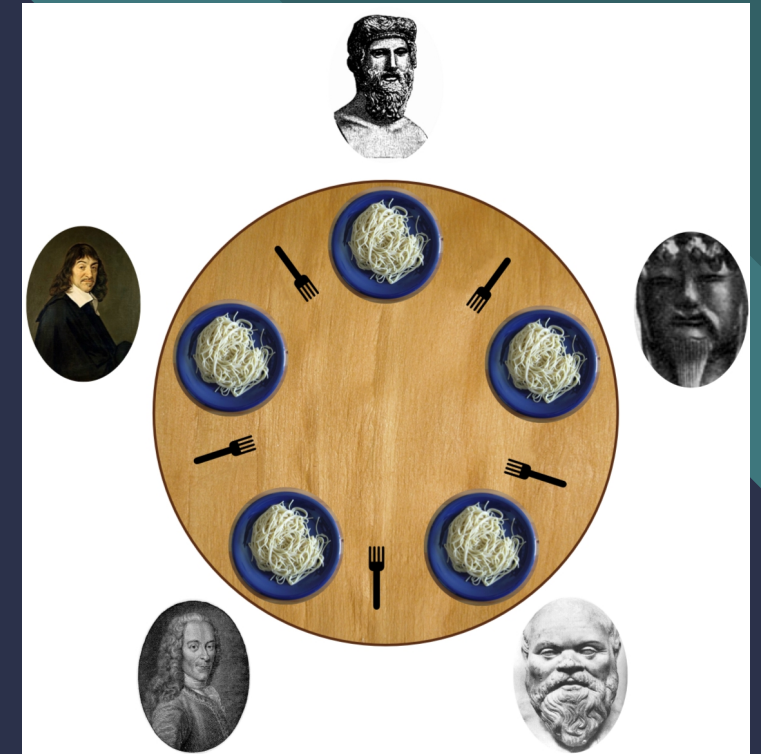
# Ejemplos. Lectores - Escritores

- El problema de **Lectores – Escritores** se produce si varios procesos pueden compartir un archivo o registro de datos.
- El proceso de sólo lectura es el "proceso de lectura" y los otros procesos son el "proceso de escritura".
- Se permite que varios objetos lean un objeto compartido al mismo tiempo pues no hay cambios en el fichero o recurso.
- Pero un proceso de escritura y otros procesos de lectura no pueden compartir el objeto al mismo tiempo.



# Ejemplos. Filósofos comensales

- Cinco **filósofos** se sientan alrededor de una mesa y pasan su vida cenando y pensando. Cada filósofo tiene un plato de fideos y un tenedor a la izquierda de su plato.
- Para comer los fideos son necesarios dos tenedores y cada filósofo sólo puede tomar los que están a su izquierda y derecha. Si cualquier filósofo toma un tenedor y el otro está ocupado, se quedará esperando, con el tenedor en la mano, hasta que pueda tomar el otro tenedor, para luego empezar a comer.
- Si dos filósofos adyacentes intentan tomar el mismo tenedor a una vez, se produce una condición de carrera: ambos compiten por tomar el mismo tenedor, y uno de ellos se queda sin comer.
- Si todos los filósofos toman el tenedor que está a su derecha al mismo tiempo, entonces todos se quedarán esperando eternamente, porque alguien debe liberar el tenedor que les falta. Nadie lo hará porque todos se encuentran en la misma situación (esperando que alguno deje sus tenedores). Entonces los filósofos se morirán de hambre. Este bloqueo mutuo se denomina **interbloqueo o deadlock**.
- El problema consiste en encontrar un algoritmo que permita que los filósofos nunca se mueran de hambre.



# Ejemplos. El barbero durmiente

- El problema consiste en una barbería en la que trabaja **un barbero** que tiene un único sillón de barbero y varias sillas para esperar.
- Cuando no hay **clientes**, el barbero se sienta en una silla y se duerme.
- Cuando llega un nuevo cliente, éste o bien despierta al barbero o —si el barbero está afeitando a otro cliente— se sienta en una silla (o se va si todas las sillas están ocupadas por clientes esperando).
- El problema consiste en realizar la actividad del barbero sin que ocurran condiciones de carrera. La solución implica el uso de semáforos y objetos de exclusión mutua para proteger la sección crítica.



# Ejemplos. Los fumadores

- El caso de **los fumadores** consiste en un grupo de fumadores que para fumar necesitan los ingredientes que les faltan para hacer un cigarrillo y fumárselo, poseen un ingrediente en cantidades ilimitadas pero les faltan otros dos. El agente posee cantidades ilimitadas de todos los ingredientes que son papel, tabaco y cerillas pero solo deja en una mesa dos de estos ingredientes a la vez. Cada fumador posee un ingrediente distinto de los tres necesarios y según los ingredientes que deje el agente uno de los fumadores podrá fumar con los dos ingredientes que el agente deja.
- El agente y los fumadores representan en la realidad a procesos y los ingredientes a los recursos de un sistema. La dificultad radica en sincronizar los agentes y fumadores para que el agente cuando deje ingredientes en la mesa el fumador correcto fume cuando.





“

"Programar sin una arquitectura o diseño en mente es como explorar una gruta sólo con una linterna: no sabes dónde estás, dónde has estado ni hacia dónde vas"

- Danny Thorpe



”

# Recursos

- Twitter: <https://twitter.com/joseluisgonsan>
- GitHub: <https://github.com/joseluisgs>
- Web: <https://joseluisgs.github.io>
- Discord: <https://discord.gg/kyhx7kGN>
- Aula Virtual:  
<https://aulavirtual33.educa.madrid.org/ies.luisvives.leganes/course/view.php?id=248>





# Gracias

José Luis González Sánchez

