TEMA 2. PROGRAMACIÓN MULTIHILO

PROGRAMACIÓN DE SERVICIOS Y PROCESOS.

1. Introducción

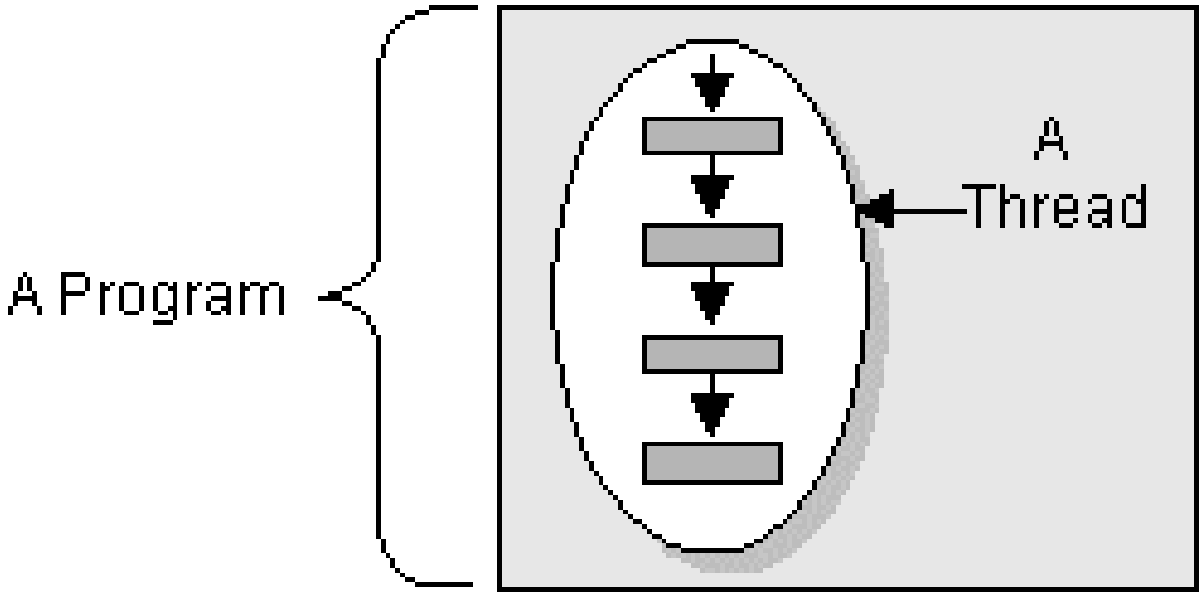
- La programación multihilo permite crear diferentes hilos de ejecución a la vez, es decir, permite realizar diferentes tareas en una aplicación de forma concurrente.
- Son multihilo la mayoría de las aplicaciones que usamos en nuestros ordenadores:
  - Editores de texto.
  - Navegadores
  - Editores gráficos
  - Etc.

2. Conceptos básicos sobre hilos

- **Hilo**: son la unidad básica de utilización de la CPU, y más concretamente de un core del procesador. Así un thread se puede definir como la secuencia de código que está en ejecución, pero dentro del contexto de un proceso.

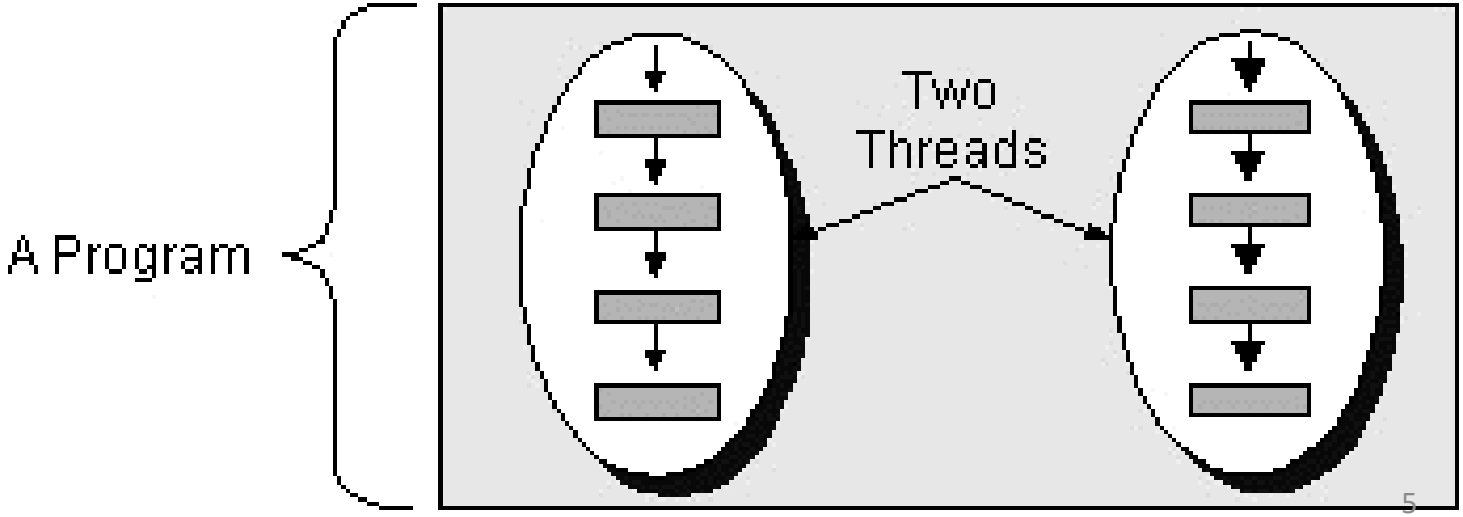
2. Conceptos básicos sobre hilos

- Los hilos representan exclusivamente las ejecuciones de las instrucciones de un programa que se llevan a cabo **simultáneamente** (concurrentemente) en el contexto de un mismo proceso. Es decir, compartiendo el acceso a la misma zona de memoria asignada al proceso al que pertenecen.



2.Conceptos básicos sobre hilos

- Dos hilos pueden ejecutarse a la vez. Dos hilos concurrentes están en progreso, o intentando obtener tiempo de ejecución de la CPU al mismo tiempo, pero no necesariamente de forma simultánea.



2.1 Recursos compartidos por hilos

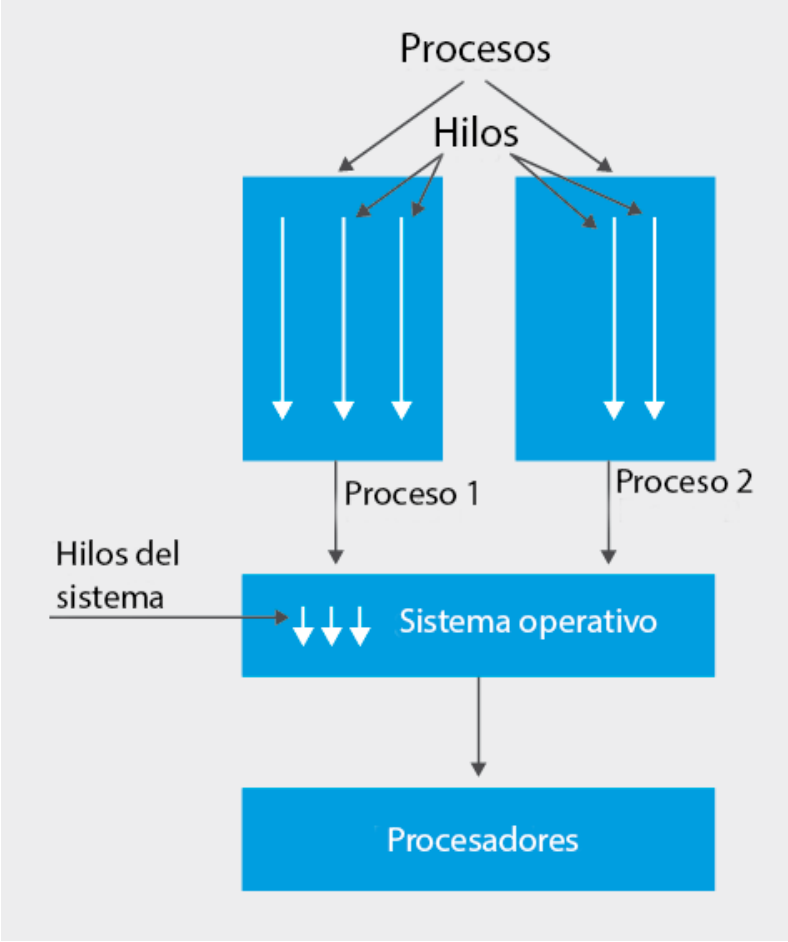
- Un hilo comparte con otros hilos:
  - La sección de código
  - Datos
  - Memoria
  - Archivos abiertos
- Cada hilo tiene:
  - Contador de programa
  - Conjunto de registros de CPU
  - Pila

2.2 Relación entre procesos e hilos

- El sistema operativo puede mantener varios procesos a la vez.
- Además, **dentro de cada proceso** pueden ejecutarse varios hilos.
  - **Es decir, bloques de instrucciones que se ejecutan en paralelo dentro del mismo proceso.**
- Procesos = **Entidades pesadas**
- Hilos = **Entidades ligeras**

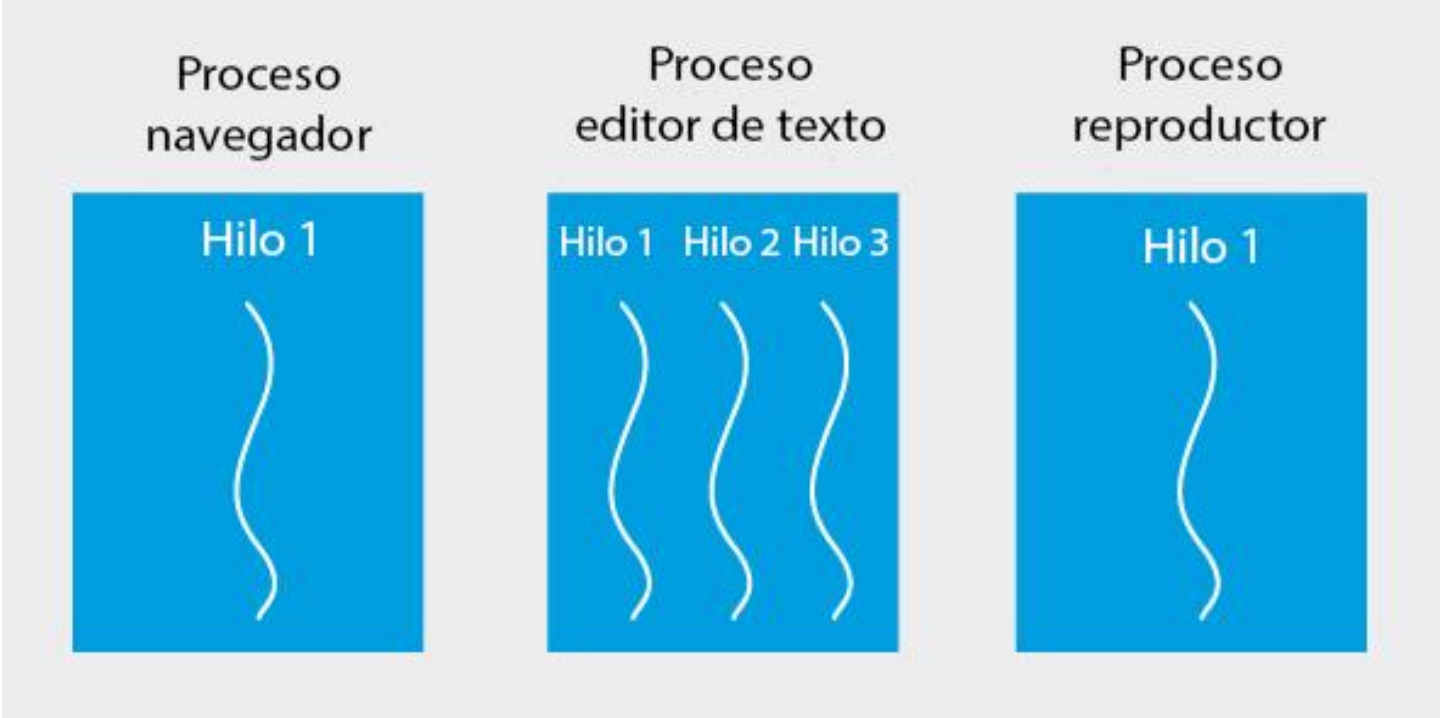
2.2 Relación entre procesos e hilos

- Cada hilo se procesa de forma independiente.



2.2 Relación entre procesos e hilos

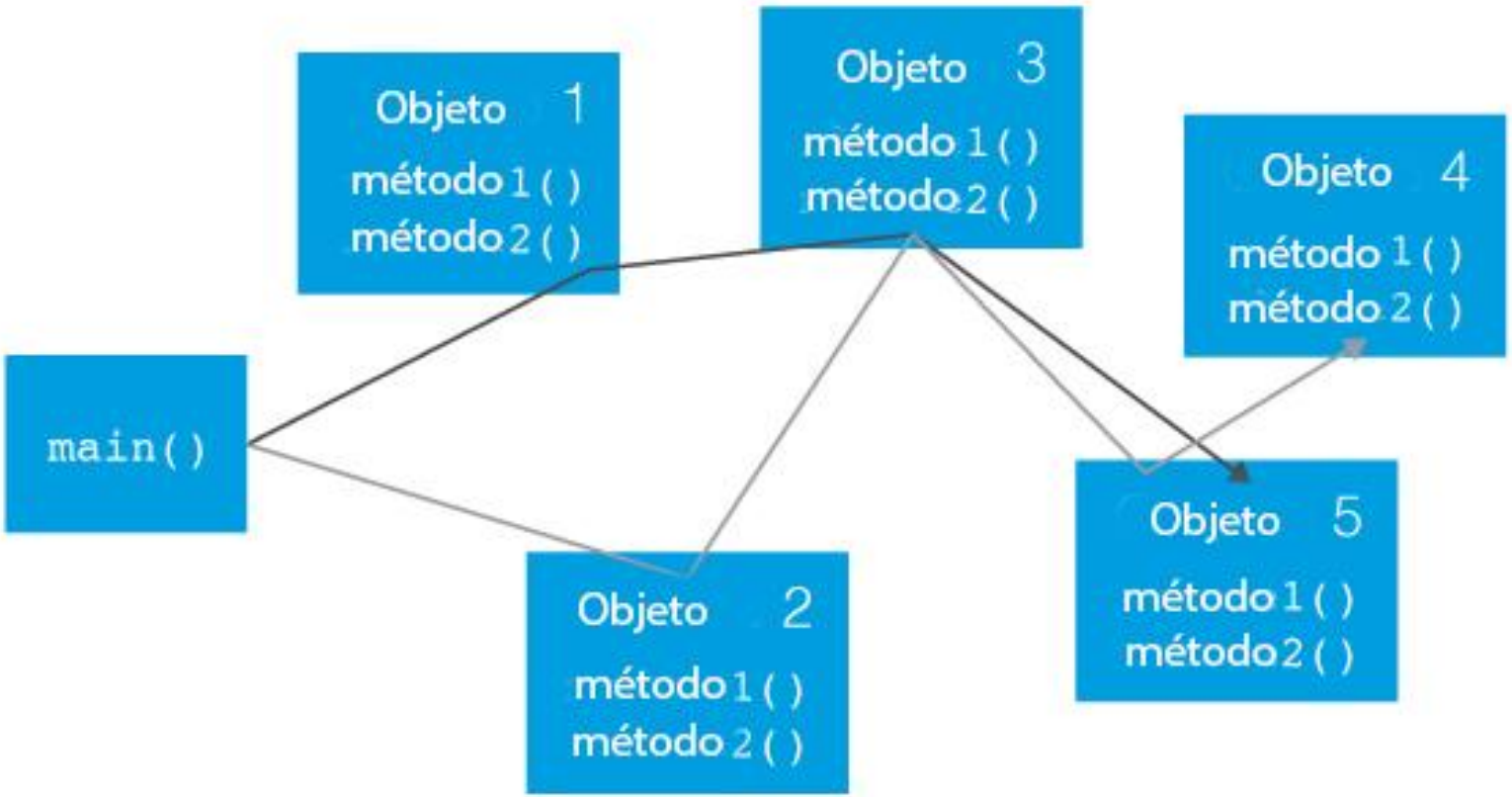
- Los sistemas operativos actuales soportan concurrencia



3. Multihilo en Java. Librerías y clases

- Cualquier programa a ejecutarse es un proceso tiene **un hilo de ejecución principal**.
- Este hilo puede a su vez crear nuevos hilos.
- Para crear hilos se puede hacer de dos modos:
  - Extendiendo de la clase Thread.
  - Implementando la interfaz Runnable. Suele utilizar una instancia a la clase Thread.

3. Multihilo en Java. Librerías y clases



3.1 Elementos relacionados con la programación de hilos

- La clase **Thread**
  - **Constructores:**
    - Thread()
    - Thread (Runnable target)
    - Thread (String name)
  - **Métodos más importantes:**
    - static Thread currentThread()
    - String getName() y void setName()
    - int getPriority() y void setPriority()
    - boolean isAlive()
    - void join()
    - void run()
    - void start()
    - static void yield()
    - static void sleep()

3.2 Creación de hilos

- Existen dos formas de creación de hilos en Java:
  - Heredando de la clase Thread

```
public class SimpleThread extends Thread{
```

  - Implementando la interfaz Runnable

```
public class SimpleThreadRunnable implements Runnable{
```

3.3 Clase Thread

- Cada hilo de ejecución de nuestras aplicaciones se asocia a una instancia de **Thread**.
- Sus métodos básicos son:
  - ***public void run():*** contiene el código que queremos ejecutar en el hilo. NO se debe invocar nunca directamente.
  - ***Public void start():*** lanza la ejecución del hilo.

3.3 Clase Thread

- Al llamar a *start()* el hilo se registra en el planificador de hilos de Java.
- Este decide que hilo ha de ejecutarse en cada momento.
- El planificador suele utilizar política FIFO.
- *start* no significa que se ejecute el hilo, si no que lo pone en estado preparado-para-ejecución.
- Cada hilo compite para entrar a ejecutarse.
- A priori, no se sabe que hilo va a ejecutarse antes.

3.4 Clase Runnable

- Si nuestra clase ya hereda de una, no puede heredar de *Thread*.
- Runnable es una interfaz que nos permite crear tareas para ser ejecutadas en hilos secundarios.
- *Thread* tiene un constructor que permite pasar como argumento un *Runnable*.

```
public interface Runnable {
    public void run();
}
```

3.4 Clase Runnable

```
public class HelloRunnable implements
Runnable {
   public void run() {
      System.out.println(“Hello from a thread!);
   }
    public static void main (String args[]) {
      (new Thread(new HelloRunnable())).start();
     }
}
```

3.5 Clase Callable<V>

- Runnable (o Thread) no permiten devolver valores.
- Complejos mecanismos de sincronización para hacerlo.
- Callable<V> básicamente es un Runnable que devuelve un valor.

```
public interface Callable<V> {
        public V call();
}
```

3.6 Clase Future<V>

- Interfaz que representa el resultado de una computación asíncrona.
- Nos permite algunas operaciones: comprobar el resultado, saber si la computación ha terminado, esperar a que termine, …
- Método **get** para obtener el valor de la ejecución de un **Callable<V>.**
- Nos “invita” a usar Executor.

3.7 Executors

- Para grandes aplicaciones, hay que separar la administración de hilos del resto de la aplicación.
- Para ello se utiliza los ejecutores o *Executors.*
- Existen tres tipos de ejecutores que son interfáces (cada una hereda de la anterior):
  - Executor: soporta lanzamiento de tareas bajo demanda:
  - **ExecutorService: añade características para administrar ciclo de vida.**
  - scheduledExecutorService: añade posibilidad de ejecutar tareas periódicas.

3.7.1 ExecutorService

- ***submit(…):*** acepta **Runnable** o **Callable.**
- Métodos para la finalización del propio ejecutor.
- Creación a partir de un pool de hilos que haga uso de *worker threads:* hilos que son reutilizables, minimizando la sobrecarga de la creación de hilos nuevos.
- Podemos finalizar el ejecutor con el método **shutdown**.

3.7.2 Pool de hilos

- Un conjunto de hilos que estarán siempre dispuestos a ejecutar tareas de tipo Runnable o Callable.
- Pueden ser de tres tipos:
  - **Single**: un solo hilo disponible. Si le pedimos (*submit*) más de una tarea a la vez, las pone en cola.
  - **Fixed**: se indica en el momento de creación el número de hilos. Si hay más tareas que hilos las pone en cola.
  - **Cached**: Crea hilos conforme enviamos tareas. Reutiliza hilos cuyas tareas han terminado, para ejecutar nuevas tareas.

3.7.3 Creación de Pool de Hilos

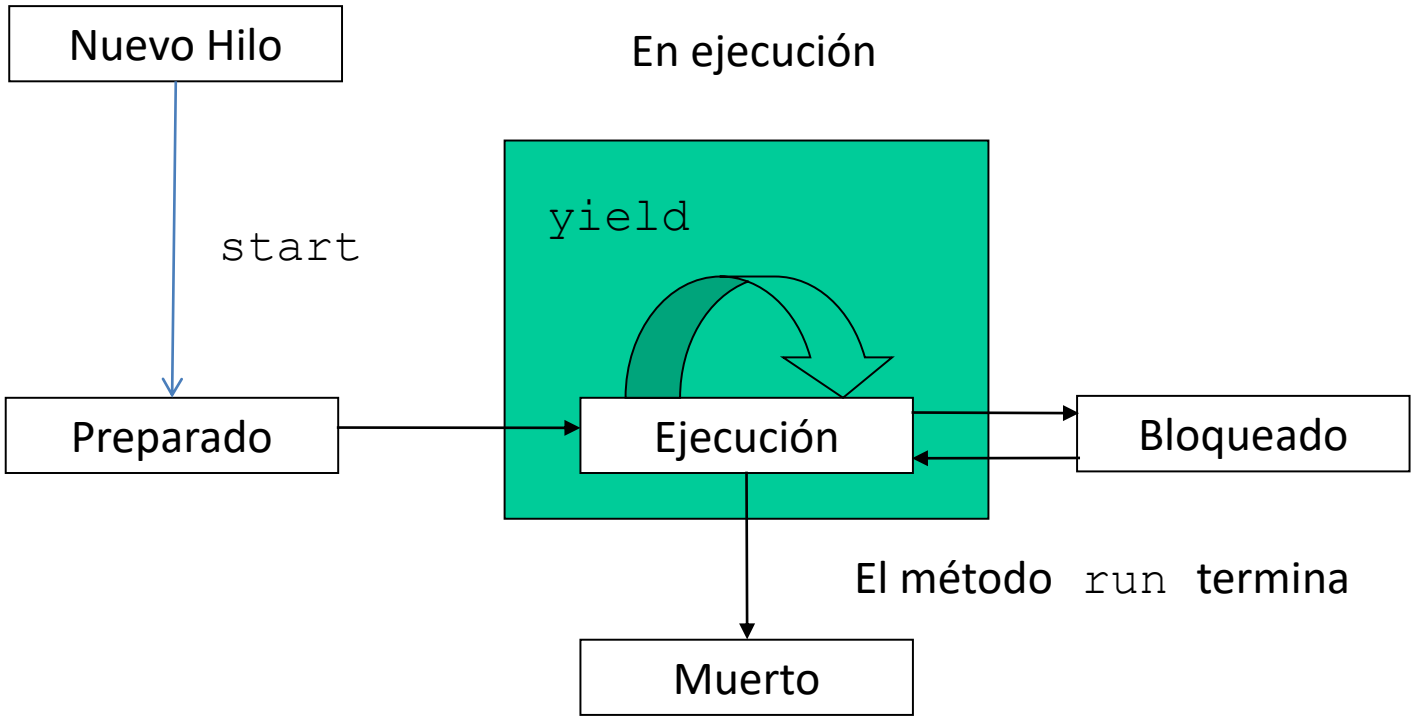
- La clase Executors tiene métopdos estáticos para construir cada tipo. Entre ellos

  - *newsingleThreadExecutor():* crea un ejecutor de tipo single.
  - *newFixedThreadPool(int n):* crea un ejecutor de tipo fixed con n hilos disponibles.
  - *newCachedThreadPool():* crea un ejecutor de tipo *cached*.

3.8.- Estados de un hilo

- Al invocar a start() pasa a **preparado para ejecutarse.**
- Al entrar en la CPU, pasa al estado **ejecución**.
- Estado **Nuevo**: el hilo se ha creado, pero no se ha invocado a start().
- **Bloqueado**: el hilo no se ejecuta porque está a la espera de un evento. Cuando sucede el evento, pasa a preparado para ejecutarse. Se puede bloquear por dos motivos:
  - **Dormido**: se bloquea por un tiempo determinado.
  - **Esperando**: se bloquea esperando la llegada de un evento. Puede ser la recepción de un mensaje, la finalización de una tarea de E/S o acceso a un método sincronizado.
- **Muerto**: Pasa a este estado cuando finaliza su método run() o recibe un mensaje interrupt(). De este estado no puede ir a ningún otro estado.

3.8.- Estados de un hilo



4.- Gestión de los hilos.

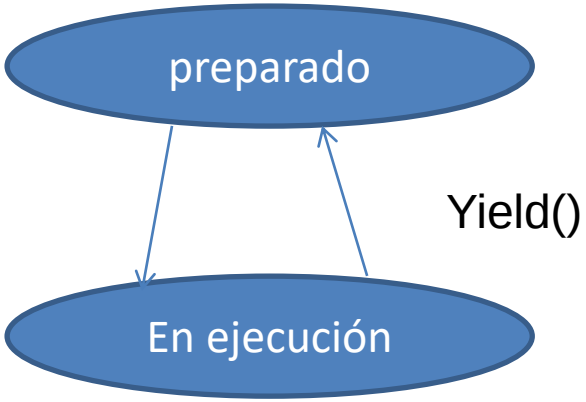
- Herramientas que incorpora Java para poder cambiar un hilo a sus distintos estados.
  - Yield
  - Dormido(sleep)
  - Join

4.1 Gestión de hilos. Yield

- Un hilo puede ofrecer su tiempo de CPU a otros hilos. Una llamada a Yield provoca que el hilo pase de En-ejecución a preparado-para-ejecución.
- Si el planificador de hilos ve que no hay ningún hilo esperando, le vuelve a poner en ejecución.

4. 1. Gestión de hilos. Yield

- Si un hilo invoca a yield, pasa a preparado-para-ejecución.
- El planificador de hilos pasará otro hilo de preparado-para-ejecución a en-ejecución.



Vemos el ejemplo:
UsoYield

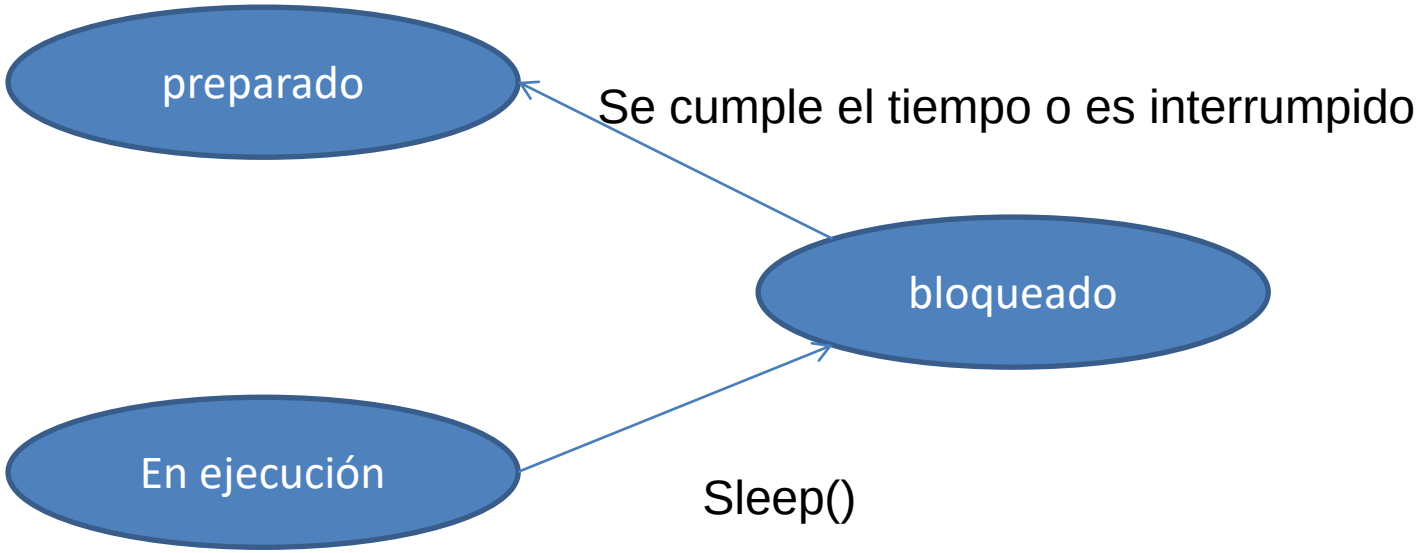
4.1. Gestión de hilos. Dormido (sleep)

- Al utilizar sleep, pasa al estado bloqueado por un determinado espacio de tiempo.
- El método sleep se define como:

```
Public static void sleep (long miliseg)
  throws InterruptedException
```

- Se bloquea por los miliseg.
- Si al estar dormido, recibe un método interrupt(), pasa a estado de preparado-para-ejecución. Despues, cuando va a En-ejecución ejecuta el código del manejador de la excepción.

4.1 Gestión de hilos. Dormido (sleep)

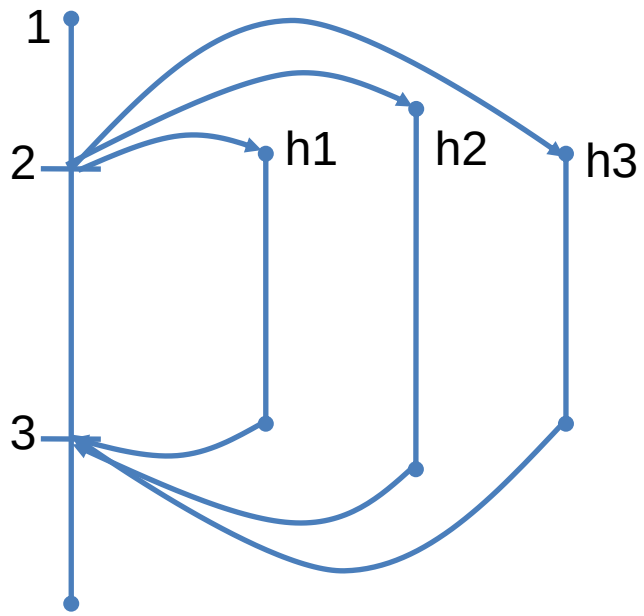


Vemos el ejemplo UsoSleep

4.1 Gestión de hilos. Join

- Un hilo invoca a join() de otro hilo para quedarse bloqueado a la espera de que termine ese segundo hilo.
- Sirve para crear estructuras de hilos.
- Puede existir un hilo padre que cree unos cuantos hijos, los arranque y espere a que estos terminen.
- Un hilo bloqueado por join, puede recibir un mensaje interrupt(), por lo que hay que tratar la excepción InterruptedException.

4.1 Gestión de hilos. Join



1. h1=new Thread()
   h2=new Thread()
   h3=new Thread()

2. h1.start()
   h2.start()
   h3.start()

3. h1.join()
   h2.join()
   h3.join()

Ejemplo de un programa principal que crea tres hilos(1), invoca el método start() de los tres hilos(2) y se queda bloqueado hasta que terminen los hilos con join()(3).

4.1 Gestión de hilos. Join

- Veremos el código de un programa que crea cuatro hilos, los ejecuta y espera a que estos terminen.
- Los hilos escriben cinco veces el número de iteración del bucle y su nombre.
- En cada iteración, después de escribir su nombre, se bloquean durante un segundo, y luego vuelven a estar disponibles para su ejecución.

4.2. Gestión de hilos. isAlive

- No se puede terminar la ejecución de un hilo. Lo único que se puede hacer es asegurarse que la ejecución del método run tenga alguna manera de finalizar.
- A veces es útil saber cuándo ha terminado un hilo.
- Para hacerlo, podemos usar el método
  - **final boolean isAlive()**

4.3 Gestión de hilos. Prioridades de los hilos

- Se puede asignar prioridad a los hilos.
- Los hilos con más alta prioridad recibirán con probabilidad más tiempo de CPU.
  - El tiempo de CPU asignado a los hilos también depende de cómo tenga implementada la multitarea el **Sistema Operativo**.
- Se puede cambiar la prioridad con el método
  - final void setPriority(int nivel)
  - Rango: **MIN_PRIORITY Y MAX_PRIORITY**

5.1 Sincronización de hilos. Compartición de información

- Necesidad de los hilos de compartir datos.
  - Las variables compartidas sirven para comunicar hilos.
  - También, los hilos pueden compartir datos de un objeto en común.
- Si no se controlan los accesos pueden darse situaciones inesperadas o erróneas.
- Es necesaria una **sincronización**.

5.2 Sincronización de hilos. Definiciones

- Condición de competencia: Un hilo entra en competencia con otro cuando ambos necesitan el mismo recurso (de forma exclusiva o no); de forma que necesitan mecanismos de sincronización y comunicación entre ellos.
- Región crítica: Es el conjunto de instrucciones de un hilo que deben ejecutarse de forma exclusiva porque el hilo utiliza un recurso que no debe usar ningún otro proceso hasta que termine.
- Condición de sincronización: Necesidad de coordinar los hilos para poder sincronizar las actividades.

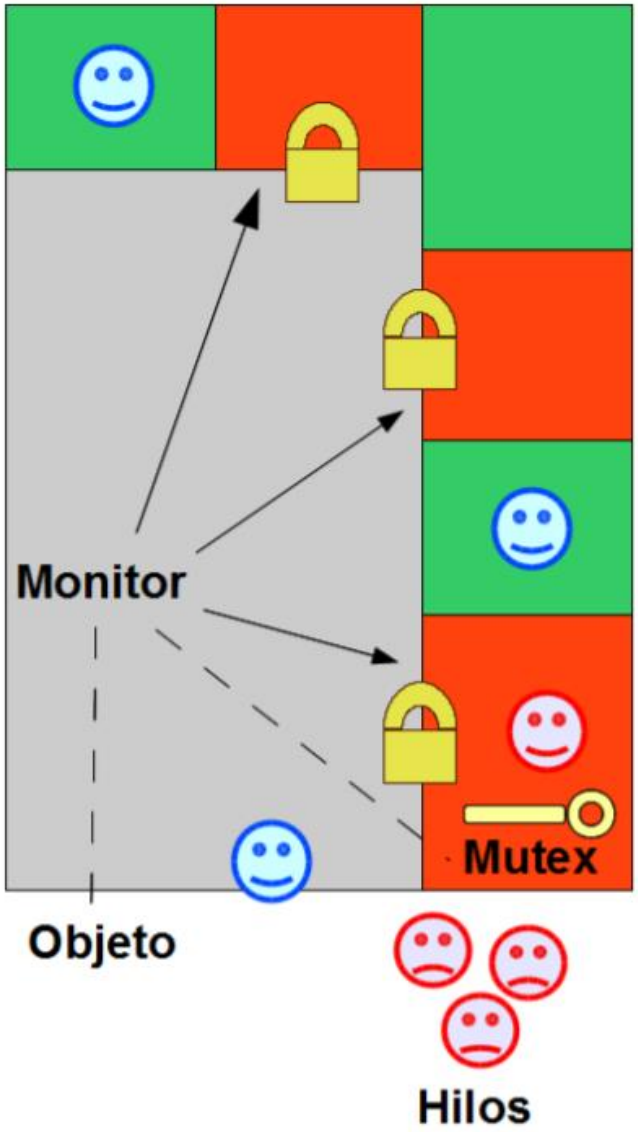
5.2 Sincronización de hilos. Definiciones

```
Entrada Sección Critica /* Solicitud per ejecutar Sección
Crítica */

/* cócigo Sección Crítica */

Salida Sección Critica /* otro proceso puede ejecutar la Sección
Crítica */
```

5.3 Monitores

- Modificador **synchronized** para conseguir la exclusión mutua.
- Nos garantiza que ningún otro método sincronizado del objeto podrá ejecutarse.
- Cuando un hilo ejecuta un método sincronizado coge el bloqueo del objeto hasta que termina su ejecución y libera el bloqueo.
- El objeto que controla los hilos que acceden a él mediante estos mecanismos se llama **monitor**.

5.3 Monitores



5.3 Monitores

- Cuando marcamos un bloque de código Java con la palabra synchronized (método o segmento de código), estamos creando un monitor asociado al objeto.

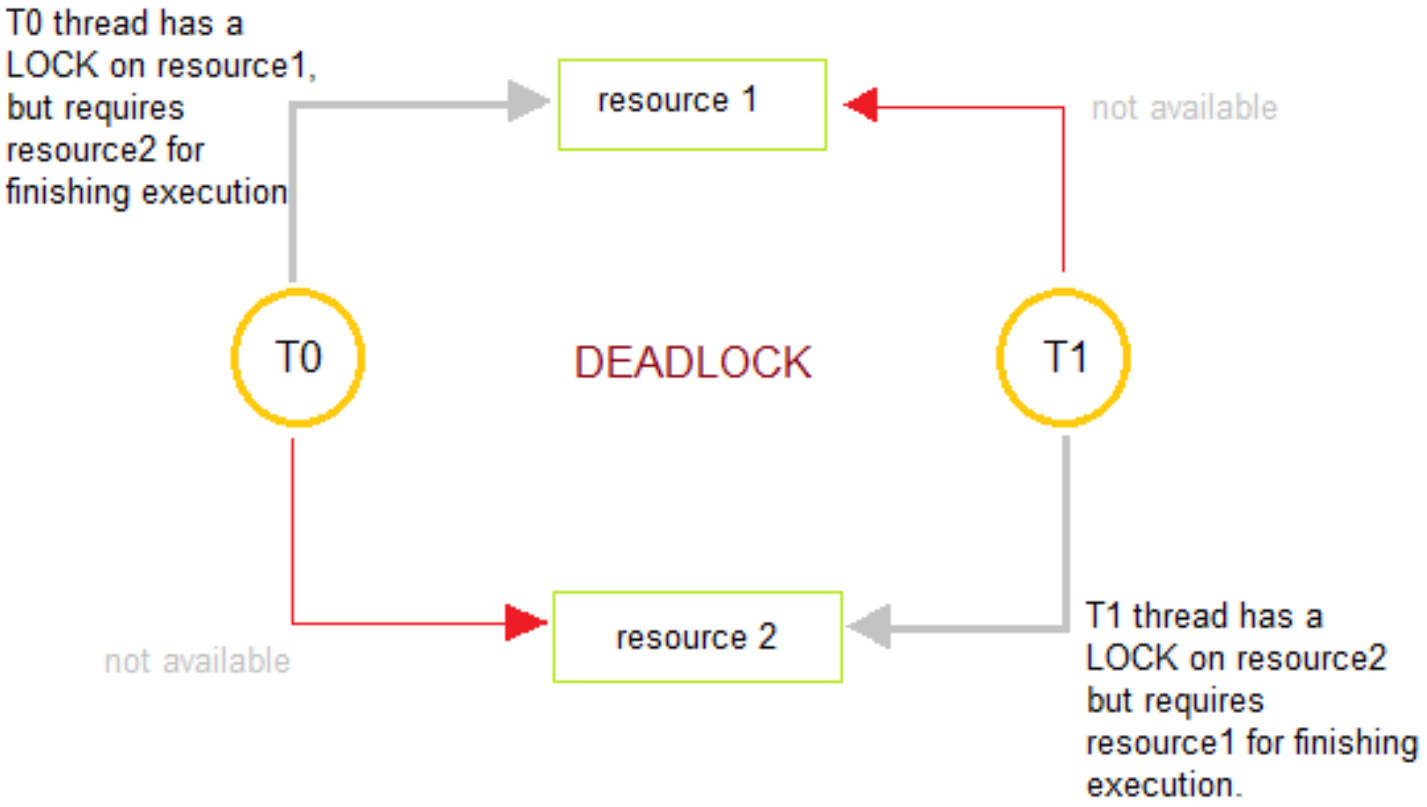
sintaxis para declarar **un método** synchronized	sintaxis para declarar **un segmento de código** synchronized

public synchronized tipodato metodo()
{
//sentencias ;
}
	synchronized (objeto)
	{
	//sentencias;
	}

5.4 Condiciones de sincronización

- Cualquier mecanismo de sincronización debe cumplir los siguientes criterios:
  - Exclusión mutua: no puede haber más de un hilo simultáneamente en la sección crítica.
  - No inanición: un hilo no puede esperar un tiempo indefinido para entrar a ejecutar la sección crítica.
  - No interbloqueo (**deadlock**): ningún proceso de fuera de la sección crítica puede impedir que otro hilo entre en la sección crítica.
  - Independencia del hardware: inicialmente no se deben hacer suposiciones respecto al número de procesadores o la velocidad de los procesos.

5.4 Condiciones de sincronización (deadlock)



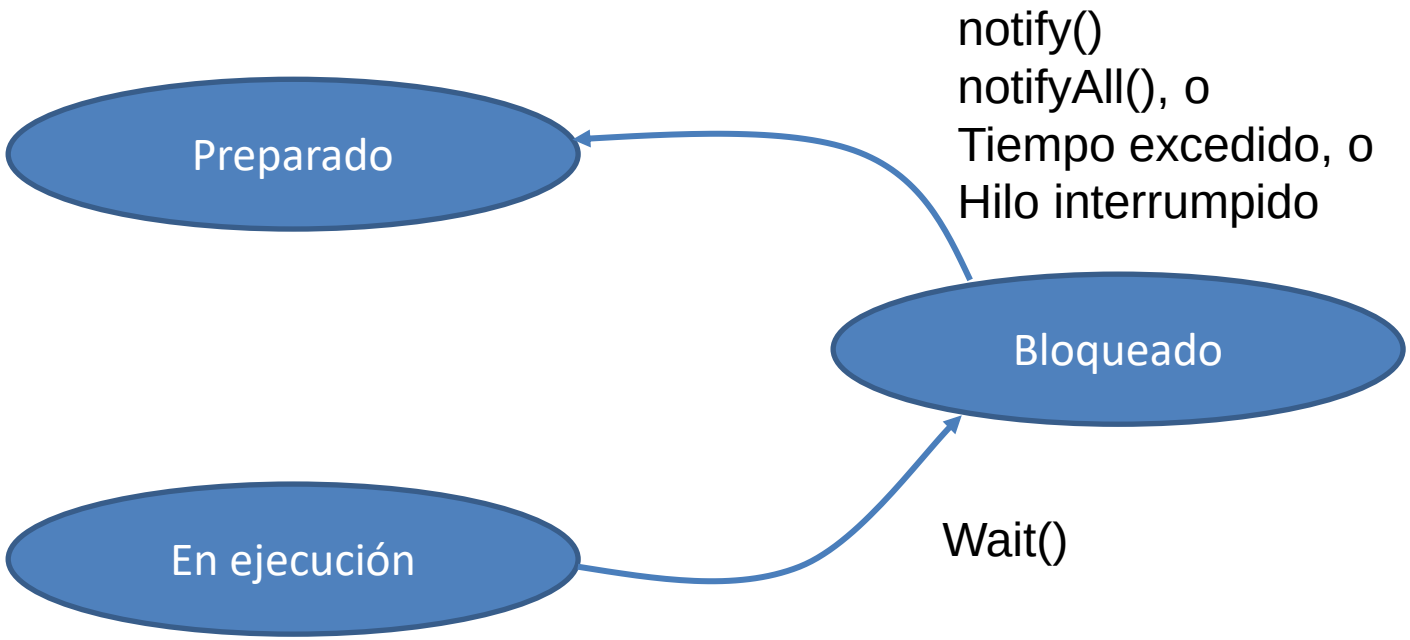
5.5 Objetos de sincronización

- Con regiones críticas se permite que varios hilos realicen la misma operación de forma sincronizada.
- Pero si queremos que varios hilos lleven a cabo distintas operaciones sobre un objeto de forma sincronizada y colaborativa, hay que usar otros métodos.

5.5 Objetos de sincronización

- Con wait(), notify() y notifyAll() seremos capaces de limitar el acceso a un objeto cuando no se den las condiciones de seguir ejecutándose, y permitir su ejecución cuando las condiciones se cumplan.
- El objeto que controla los hilos que acceden a él mediante estos mecanismos se llama **monitor**.

5.5 Objetos de sincronización



Cuando un hilo invoca a wait(), queda bloqueado a la espera de un evento. Puede ser la invocación de notify, notifyAll, superar el tiempo especificado en Wait o la interrupción del hilo.

5.5 Productor-consumidor

- Veremos estos métodos con un problema de sincronización típico: productor y consumidor.

- Esquema de aplicación productores-consumidores.

5.5 Productor-consumidor

- DESCRIPCION DEL EJERCICIO. PRODUCTORES
- En nuestro esquema hay uno o varios hilos productores, encargados de llenar un recipiente.
- El productor llenará el recipiente siempre que este vacio. Si está lleno, debe bloquearse.
- El productor saldrá del estado de bloqueo cuando se le notifique que el recipiente está vacio.

5.5 Productor-consumidor

- El productor debe notificar a los hilos consumidores que están bloqueados en espera de que se llene el recipiente, que este se ha llenado.

```
SI recipiente_lleno ENTONCES
   Esperar_vaciar_recipiente
EN OTRO CASO
   Llenar_recipiente
Notificar_llenado_Recipiente
```

5.5 Productor-consumidor

- DESCRIPCION DEL EJERCICIO. CONSUMIDORES
- El consumidor vaciará el recipiente si está lleno.
- Si no está lleno, se bloqueará a la espera de que se le notifique que se ha llenado.
- Una vez vacio el recipiente, debe notificar a los hilos que estaban esperando para llenar el recipiente, que este ha sido vaciado.

5.5 Productor-consumidor

```
SI recipiente_vacio ENTONCES
   Esperar_llenar_recipiente
EN OTRO CASO
   Vaciar_recipiente
Notificar_vaciado_Recipiente
```

- El acceso al objeto compartido, recipiente, ha de hacerse de forma sincronizada, usando regiones críticas.

5.6 Semáforos

- Un semáforo binario es un indicador de condición que registra si un recurso está en uso o no.
- Un semáforo binario sólo puede tomar dos valores: 0 y 1.
- Si 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.

5.6 Semáforos

- 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 o acquire)
  - Señal (signal o release)

5.6 Semáforos

- Para **crear y usar un objeto** Semaphore haremos lo siguiente:

1. Indicar al constructor Semaphore (int permisos) el total de permisos que se pueden dar para acceder al mismo tiempo al recurso compartido. Este valor coincide con el número de hilos que pueden acceder a la vez al recurso.

5.6 Semáforos

2. Indicar al semáforo mediante el método acquire(), que queremos acceder al recurso, o bien mediante acquire(int permisosAdquirir) cuántos permisos se quieren consumir al mismo tiempo.
3. Indicar al semáforo mediante el método release(), que libere el permiso, o bien mediante release(int permisosLiberar), cuantos permisos se quieren liberar al mismo tiempo.

5.6 Semáforos

- Hay otro constructor Semaphore (int permisos, boolean justo) que mediante el parámetro justo permite garantizar que el primer hilo en invocar adquire() será el primero en adquirir un permiso cuando sea liberado. Esto es, garantiza el orden de adquisición de permisos, según el orden en que se solicitan.

6. Programación de aplicaciones multihilo

- Consejos para la programación de aplicaciones multihilo
  - Identificar qué tareas se pueden realizar en paralelo y cuáles no.
  - Identificar los recursos compartidos por varios hilos.
  - Establecer las regiones críticas. Es decir, los trozos de código que acceden de manera simultánea a un recurso compartido.
  - Establecer la exclusión mutua en las regiones críticas utilizando monitores o semáforos.
  - Sólo establecer la exclusión mutua donde sea necesario. No hay que abusar de este mecanismo, ya que, estaremos perdiendo capacidad de concurrencia.
  - No suponer el orden de ejecución de las instrucciones ejecutadas por los hilos.
  - Establecer mecanismos de sincronización allí donde sea necesario.