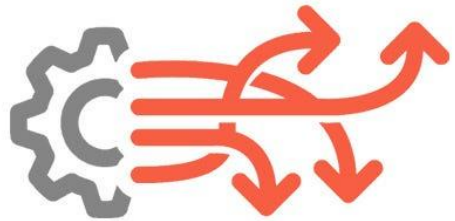
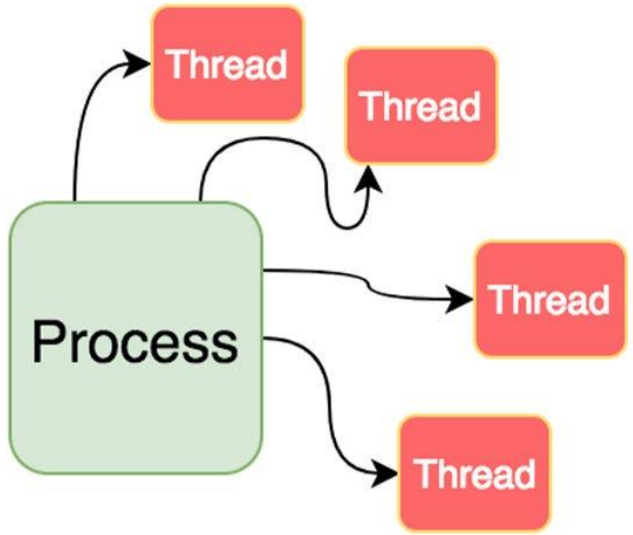
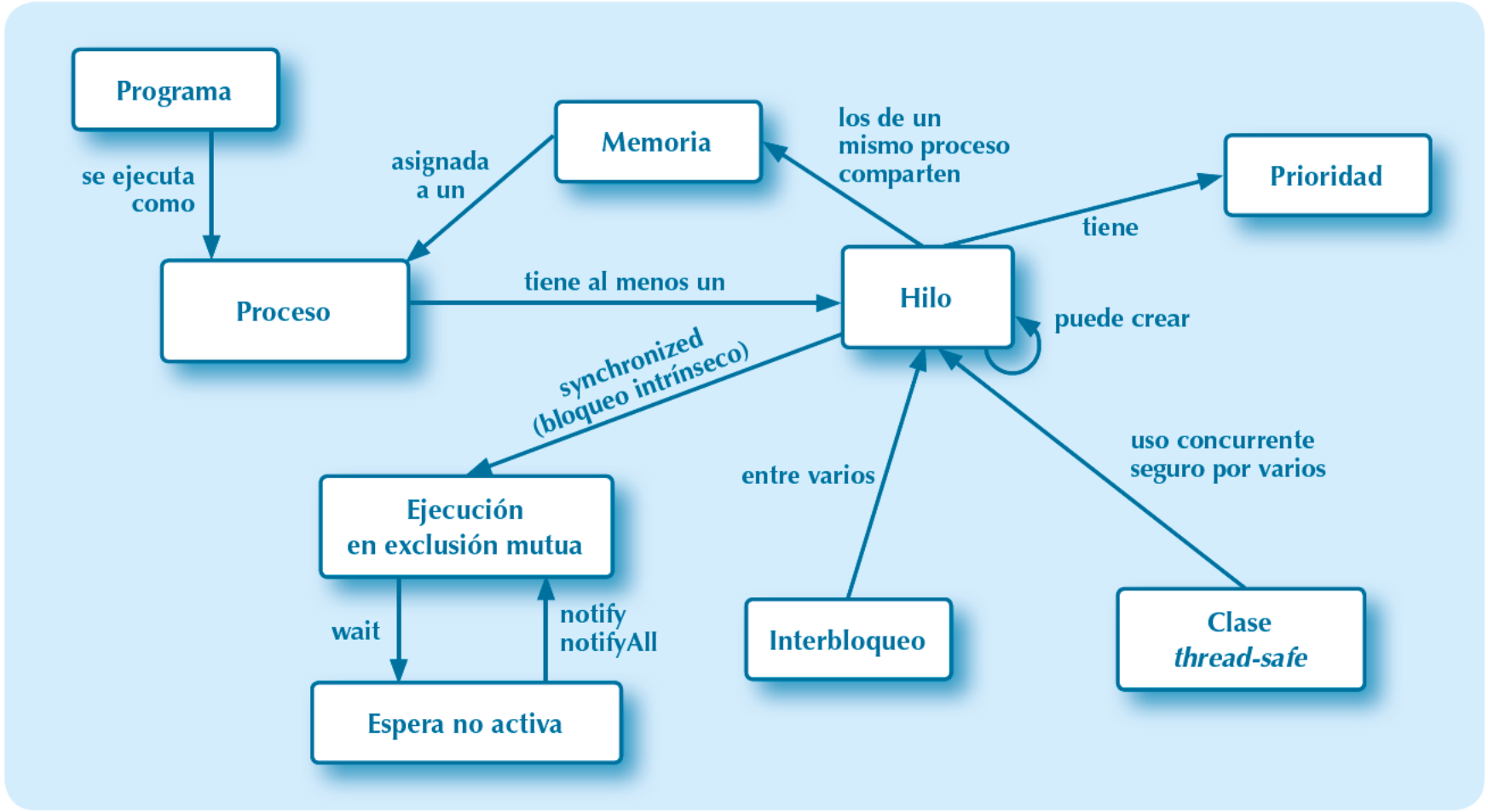
Unidad Temática 02

Programación de Hilos





Mapa conceptual

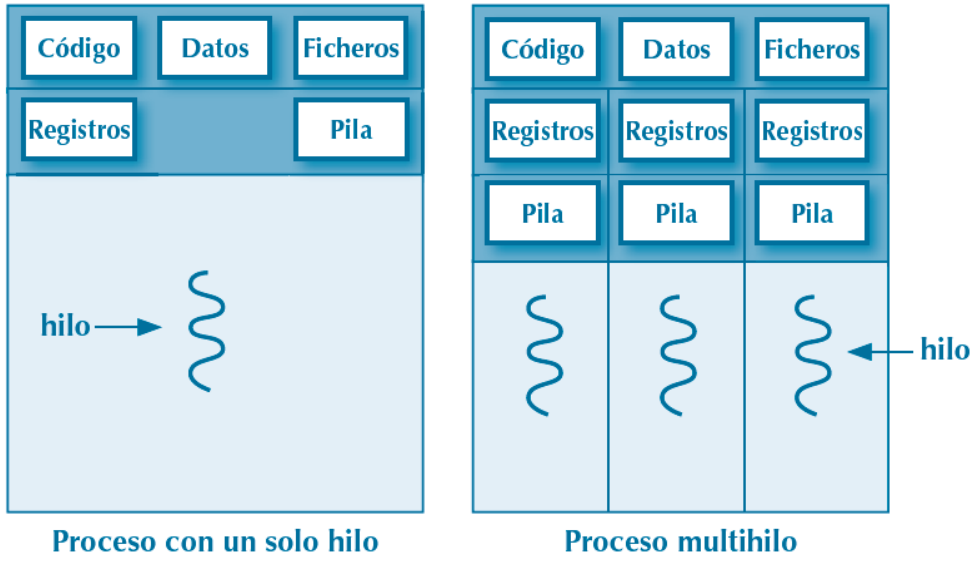


¿Qué es un hilo?

- Es una **unidad de ejecución** dentro de un programa.
- Los programas pueden tener **múltiples hilos ejecutándose en paralelo**.
- La ejecución de **un proceso comienza con un único hilo**, pero puede crear otros.
- Los distintos hilos de un mismo proceso **comparten**:
  - El **espacio de memoria** asignado al proceso (inc. datos como variables globales).
  - La **información de acceso a ficheros**. No solo para almacenar datos, sino también para controlar dispositivos de entrada y salida (E/S).

¿Qué es un hilo?

- Cada hilo tiene sus propios valores para:
  - Los registros del procesador.
  - El estado de su pila *(stack).* En la pila se guarda información acerca de las llamadas en curso de ejecución a métodos de diversos objetos. Para cada llamada se guardan, entre otras cosas, los datos locales (en variables internas del método).



Ventajas

- 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.
  - Conmutación bastante **más rápida** entre hilos del mismo proceso que entre procesos.

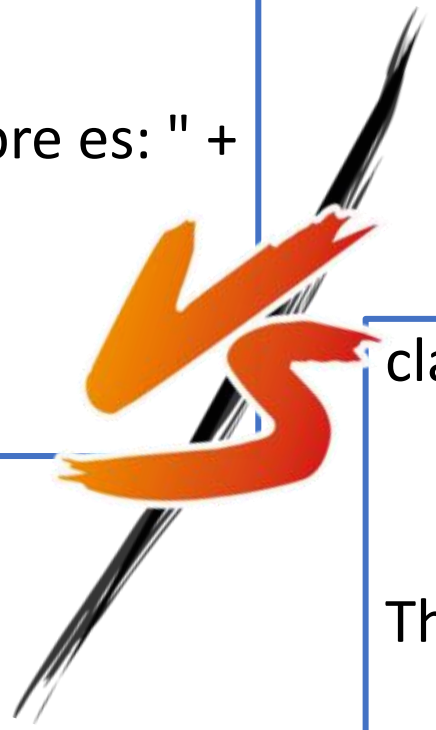
- Por esas razones, a los hilos se les denomina también procesos ligeros.

Uso

- 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.

Creación de Hilos en Java

```
class Hilo extends Thread {
    public void run() {
        System.out.println(“Mi nombre es: " +
this.getName());
    }
}
```

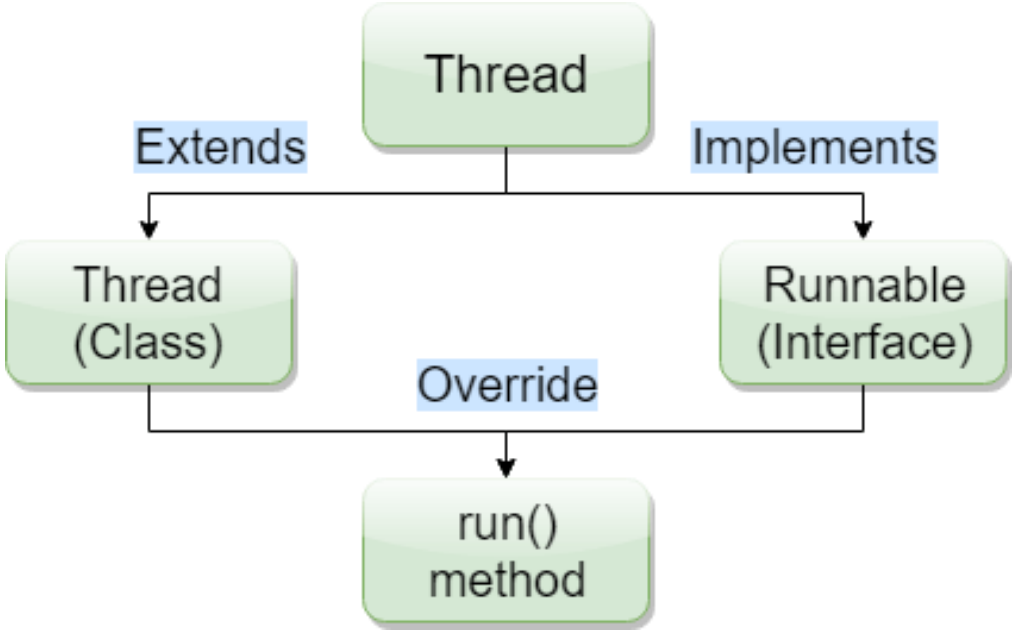


```
class Hilo implements Runnable {
    public void run() {
        System.out.println(“Mi nombre es: " +
Thread.currentThread().getName());
    }
}
```

¿Qué opción es mejor?

- Las dos opciones son, en la práctica, iguales porque al final es un *Thread* el que se ejecuta.

- Es aconsejable utilizar **Runnable** cuando necesitamos que nuestra clase herede de otra clase distinta de **Thread** (por ejemplo, la clase *Villano* que hereda de *Personaje* y necesito que ejecute como un *Thread* junto con el resto de *Personajes*).



¡Java no permite herencia múltiple!

Ejemplo práctico 1

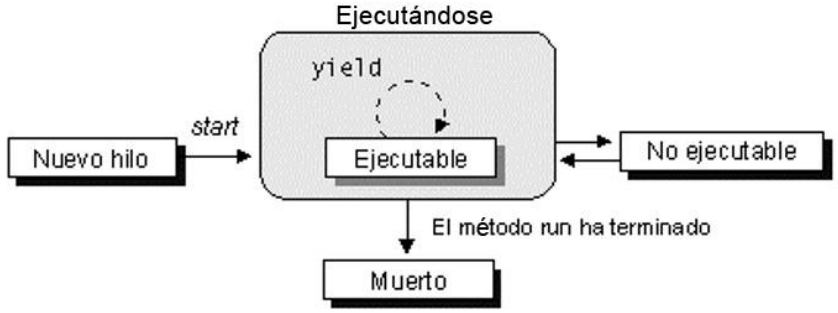
- Escribe un programa en Java que cree dos hilos, uno que extienda la clase *Thread* y otro que implemente la interfaz *Runnable*.

- El método *run()* de ambos hilos debe simplemente saludar por pantalla.

- Revisa si realmente se han creado diferentes hilos…

Thread.currentThread().getName()

Ciclo de vida de un hilo



- Un hilo en Java pasa por varios estados durante su ejecución. Estos son controlados por la JVM y las acciones realizadas sobre el hilo.

- Estados de un hilo
  - **Nuevo (NEW):** El hilo se ha creado, pero no ha iniciado → *Thread hilo = new Thread ( )*
  - **Ejecutable (RUNNABLE):** El hilo está listo para ejecutarse, pero espera que el sistema operativo le asigne tiempo de CPU → *hilo.start ( )*
  - **En ejecución (RUNNING):** El hilo está siendo ejecutado activamente.
  - **Bloqueado o en espera (BLOCKED, WAITING, TIMED_WAITING):** El hilo está esperando por un recurso o una señal → *hilo.sleep ( ) / hilo.wait ( )*
  - **Terminado (TERMINATED):** El hilo ha completado su ejecución → Fin de método *run ( )*

Métodos para gestionar hilos

Método	Descripción	Ejemplo
start()	Inicia hilo (estado *runnable*).	hilo.start()
sleep(long ms)	Pausa el hilo durante un tiempo.	Thread.sleep(500)
join()	Espera a que termine otro hilo o devuelve la ejecución al principal.	hilo.join()
setName(String name)	Cambia el nombre del hilo.	hilo.setName(“hilo1”)
setPriority(int priority)	Cambia la prioridad del hilo (MIN_PRIORITY, NORM_PRIORITY, MAX_PRIORITY)	hilo.setPriority(Thread.MAX_PRIORITY)
currentThread()	Referencia al hilo actual	Thread.currentThread().getName()

Aviso a navegantes

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

  IllegalThreadStateException

- El orden en el que inicies los hilos mediante start() no influye en el orden de ejecución de los mismos, es decir:
  - El orden de ejecución de los hilos es no determinístico
  - Se desconoce la secuencia en la que serán ejecutadas las instrucciones del programa.

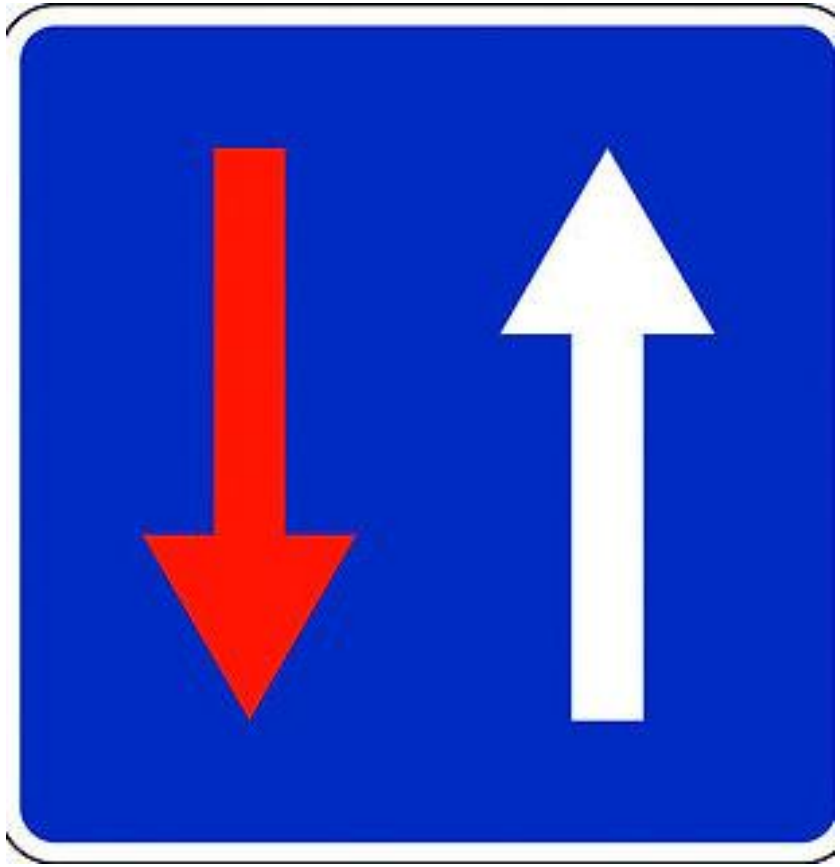
Ejemplo práctico 2

- Escribe un programa en Java que lance tres hilos, cada uno con un nombre diferente, que impriman su nombre y un contador del 1 al 5.
- Cada hilo debe pausarse durante 500 milisegundos entre cada iteración.
- El programa principal debe esperar a que terminen todos e informar cuando esto suceda.

** Una vez funcione, modifica prioridades de los hilos y prueba distintas ejecuciones.*

¿Sirve de algo asignar prioridad a los hilos?

- En la teoría…
  - Un hilo con mayor prioridad **debería tener más tiempo de CPU** que uno con menor prioridad.
  - Si hay múltiples hilos ejecutándose y los recursos son limitados, los hilos con mayor prioridad **tendrán preferencia** para ser seleccionados.
- En la práctica…
  - El sistema operativo tiene su **propio planificador** y puede ignorar a la JVM.
  - Si **sobran recursos** en la CPU, la prioridad no tiene apenas impacto.



Sincronización de hilos

- Cuando las tareas que realizan los hilos son independientes, puede dar igual el orden en que hagan las cosas.
- Sin embargo, los hilos suelen utilizarse para colaborar en tareas comunes o en un entorno común. Aquí surgen problemas de sincronización.
- Veamos ejemplo del contador colaborativo.

Sincronización de hilos

Condiciones de carrera

- ¿Qué ha pasado con el contador?

`registro ← [cuenta]`	Copia valor de la variable `cuenta` de memoria principal a `registro`
`incrementar registro`	Incrementar el valor de `registro`
`registro → [cuenta]`	Copia valor de `registro` a variable `cuenta` en memoria principal

- Cuando estas operaciones se ejecutan en un hilo, la ejecución del hilo se puede interrumpir y **se pueden intercalar entre ellas operaciones de otro hilo**.
- Según cómo se intercalen las operaciones de los distintos hilos, los resultados pueden no ser los deseados. Esto es lo que se conoce como una ***condición de carrera*** (***race condition*** en inglés).
- Existen condiciones de carrera cuando son posibles ejecuciones concurrentes de dos o más procesos que dan lugar a resultados incorrectos.

Sincronización de hilos

Condiciones de carrera

- Ejemplo de *RACE CONDITION*

Resultados incorrectos con ejecución concurrente de dos hilos

	Situación inicial: valor en la variable **cuenta** es 86	
H1	`registro ← [cuenta]`	Se guarda valor 86 en **registro** para H1
H2	`registro ← [cuenta]`	Se guarda valor 86 en **registro** para H2
H1	`incrementar registro`	El valor de **registro** pasa a ser 87 para H1
H2	`incrementar registro`	El valor de **registro** pasa a ser 87 para H2
H1	`registro → [cuenta]`	El valor en la variable **cuenta** pasa a ser 87
H2	`registro → [cuenta]`	El valor en la variable **cuenta** pasa a ser 87. Se ha perdido el incremento hecho por H1

Sincronización de hilos

Condiciones de carrera

- Solución → Bloques sincronizados para **secciones críticas de código**

```
class Contador {
  private int cuenta = 0;

  synchronized public int getCuenta() {
    return cuenta;
  }

  synchronized public int incrementa() {
    this.cuenta++;
    return cuenta;
  }
}
```

- Solo un hilo puede estar dentro de un método synchronized del mismo objeto c a la vez.
- Cuando un hilo entra en incrementar(), bloquea el monitor del objeto c.
- Mientras ese hilo no salga, ningún otro hilo puede entrar ni en incrementar() ni en decrementar().

Sincronización de hilos
Condiciones de carrera

- **Pero ojo con estos matices:**
  - Si tienes **distintas instancias** de la clase Contador, cada una tiene su **propio cerrojo**. Dos hilos podrían ejecutar métodos sincronizados al mismo tiempo, pero en **objetos diferentes**.
  - Si los métodos fueran static synchronized, el bloqueo sería **de la clase**, no del objeto. En ese caso, se bloquearía el acceso global a todos los métodos estáticos sincronizados de esa clase.
  - Si solo sincronizas uno de los dos métodos (incrementar() sí, decrementar() no), ya no hay garantía.

Sincronización de hilos
La interfaz Lock

- **Diferencia entre synchronized y Lock:**
  - **Lock** ofrece mayor flexibilidad, como intentar adquirir el bloqueo (*tryLock*) y bloquear con tiempo límite.
  - Permite un control más explícito sobre la adquisición y liberación de bloqueos.
- **Métodos principales de Lock:**
  - **lock()**: Adquirir el bloqueo.
  - **unlock()**: Liberar el bloqueo.
  - **tryLock()**: Intentar adquirir el bloqueo sin bloquear indefinidamente (por un tiempo especificado).
- **Buenas prácticas:**
  - Siempre liberar el bloqueo en un bloque *finally*.

```
Lock lock = ...;
if (lock.tryLock()) {
  try {
    // manipulate protected state
  } finally {
    lock.unlock();
  }
} else {
  // perform alternative actions
}
```

Sincronización de hilos
La interfaz Lock

```
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

private final Lock lock = new ReentrantLock();

// Método para depositar dinero
public void depositar(double cantidad) {
        lock.lock();
        try {
                saldo += cantidad;
        }
        finally {
                lock.unlock();
        }
}
```

Sincronización de hilos

Semáforos

- Un **semáforo** es una herramienta de sincronización utilizada en programación concurrente para controlar el acceso a recursos compartidos limitados.

- **Concepto clave:**
  - Un semáforo tiene un número de **permisos**.
  - Los hilos deben **adquirir un permiso** para acceder al recurso.
  - Cuando terminan, deben **liberar el permiso**.

- **Ejemplo:**
  Imagina una sala con 3 ordenadores. Solo 3 personas pueden usar los ordenadores al mismo tiempo. Si la sala está llena, las demás personas deben esperar.

```
Semaphore ordenadores = new Semaphore(3);
```

Sincronización de hilos

Semáforos

- **acquire()**
  - El hilo intenta adquirir un permiso.
  - Si no hay permisos disponibles, el hilo espera.
- **release()**
  - Libera un permiso, permitiendo que otros hilos lo adquieran.
- **tryAcquire()**
  - Intenta adquirir un permiso sin bloquearse.
  - Devuelve true si lo consigue, false en caso contrario.

Sincronización de hilos

Semáforos

- **Semáforo Binario:**
  - Tiene solo 1 permiso.
  - Similar a un candado: un hilo puede "bloquear" el recurso y otro debe esperar.
- **Semáforo Contador:**
  - Tiene varios permisos.
  - Útil para controlar un número limitado de recursos, como una cantidad fija de conexiones de red.

Sincronización de hilos

Semáforos

¿Cuándo usar semáforos?

- Controlar acceso a recursos limitados.
  - Ejemplo: Máximo de usuarios simultáneos en un sistema.
- Sincronizar hilos que deben esperar un evento.
  - Ejemplo: Esperar a que se liberen recursos.
- Evitar condiciones de carrera en escenarios complejos.
  - Ejemplo: Controlar acceso a un buffer compartido.

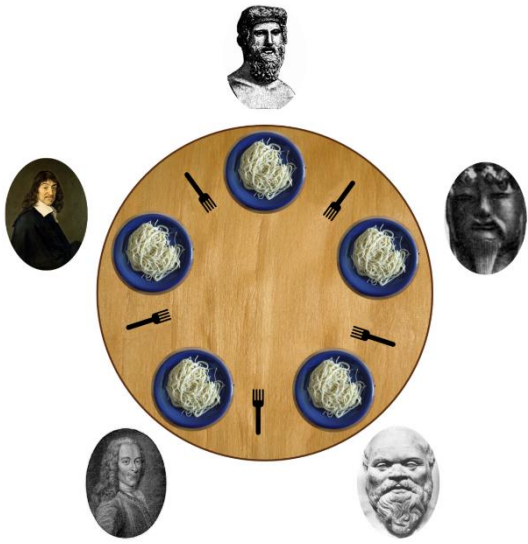
Ejemplo práctico – Cena de filósofos

- En una mesa hay procesos que simulan el comportamiento de unos **filósofos** que intentan comer de un plato. Cada filósofo tiene un **cubierto** a su izquierda y uno a su derecha y para poder comer tiene que conseguir los dos. Si lo consigue, mostrará un mensaje en pantalla que indique «*Filosofo 2 comiendo*».

- Después de comer, **soltará los cubiertos y esperará** al azar un tiempo entre 1 y 5 milisegundos, indicando por pantalla «*El filósofo 2 está pensando*».

- En general, todos los objetos de la clase Filósofo están en un **bucle infinito** dedicándose a comer y a pensar.

- Simular este problema en un programa Java que muestre el progreso de todos **sin caer en problemas de sincronización ni de inanición**.



Sincronización de hilos

Bloqueo intrínseco

- Cada objeto de Java posee un bloqueo intrínseco por el solo hecho de pertenecer a la clase *Object*.
- Cada bloqueo intrínseco tiene asociada una cola de procesos esperando a adquirirlo.
- Un hilo no puede ejecutar un bloque de código sincronizado sobre un objeto si ya existe un hilo que ha adquirido el bloqueo para dicho objeto.
- Los objetos de bloqueo deberían ser de tipo *final* porque, si se le asigna un nuevo valor, quedan sin efecto todos los bloqueos que hubiera sobre dicho objeto.

Sincronización de hilos

Bloqueo intrínseco

```
public class BloqueoIntrinseco {
    private long contador1 = 0;
    private long contador2 = 0;
    private final Object lock1 = new Object();
    private final Object lock2 = new Object();

    public void incrementar1(){
        synchronized (lock1){
            contador1++;
        }
    }

    public void incrementar2(){
        synchronized (lock2){
            contador2++;
        }
    }
}
```

Sincronización de hilos
Interbloqueo (deadlock)

- **Definición:**

Un interbloqueo ocurre cuando dos o más procesos o hilos quedan bloqueados esperando recursos que están ocupados por otros procesos, generando un ciclo de dependencia en el que nadie puede continuar. Esto ocurre en el problema de los filósofos.

- **Características:**
  - **Dependencia Circular:** Cada hilo espera por un recurso que otro hilo ya tiene.
  - **Exclusión Mutua:** Los recursos no pueden ser compartidos.
  - **Espera Mantenida:** Los hilos mantienen los recursos que ya tienen mientras esperan otros recursos.
  - **Ausencia de Expropiación:** Los recursos no pueden ser forzadamente liberados.

- **Consecuencias:**
  - Ningún hilo puede avanzar.
  - El sistema queda en un estado inactivo hasta que se intervenga manualmente.

Sincronización de hilos
Prevención del Interbloqueo

La idea principal es **diseñar el sistema de forma que no pueda ocurrir un interbloqueo**. Para ello, se rompen una o más de las cuatro condiciones necesarias para que ocurra un interbloqueo (exclusión mutua, espera mantenida, ausencia de expropiación y dependencia circular).

- **Romper la Dependencia Circular**
  - **Descripción:** Cambiar el orden en que se adquieren los recursos para evitar que los procesos formen un ciclo de espera.
  - **Ejemplo:** Hacer que el último filósofo tome los cubiertos en orden inverso (primero el derecho, luego el izquierdo). Esto asegura que no haya un ciclo de dependencia.

- **Permitir la Expropiación**
  - **Descripción:** Permitir que un recurso ya adquirido por un hilo sea expropiado (forzosamente liberado) y asignado a otro hilo si es necesario.
  - **Ejemplo:** Si un filósofo no puede conseguir el segundo cubierto en un tiempo determinado, suelta el primero y lo pone a disposición de otros filósofos.

- **Evitar la Exclusión Mutua**
  - **Descripción:** Permitir que los recursos compartidos sean utilizados simultáneamente si es posible.
  - **Ejemplo:** En sistemas reales, esto podría implicar el uso de estructuras de datos thread-safe o secciones críticas con acceso concurrente controlado.

- **Evitar la Espera Mantenida**
  - **Descripción:** Obligar a los hilos a solicitar todos los recursos que necesitan al mismo tiempo, en lugar de adquirirlos uno por uno.
  - **Ejemplo:** Cada filósofo solo toma los dos cubiertos si ambos están disponibles al mismo tiempo.

Sincronización de hilos
Detección y recuperación del Interbloqueo

Si no se puede prevenir el interbloqueo, se pueden implementar estrategias para **detectarlo cuando ocurra y recuperarse del mismo**.

- **Algoritmo de Detección**
  - **Descripción:** Implementar un algoritmo que monitoree continuamente el sistema para identificar ciclos de dependencia.
  - **Cómo funciona:**
    - Representar las solicitudes de recursos como un **grafo de asignación**.
    - Detectar ciclos en el grafo. Si hay un ciclo, hay un interbloqueo.
  - **Ejemplo:** Monitorizar qué filósofos están esperando recursos y liberar el ciclo de forma manual o automática.

- **Recuperación del Interbloqueo**
  - **Descripción:** Una vez detectado el interbloqueo, liberar recursos para romper el ciclo.
  - **Técnicas comunes:**
    - **Abortar procesos:** Terminar uno o más procesos para liberar recursos. En el problema de los filósofos, esto sería "sacar" a un filósofo de la mesa.
    - **Liberar recursos:** Obligar a uno o más procesos a liberar sus recursos y reintentarlo más tarde.

- **Ignorar el Problema (Estrategia del "Avestruz")**
  - **Descripción:** Esta estrategia es válida en sistemas donde la probabilidad de un interbloqueo es extremadamente baja o su impacto es tolerable.
  - **Ejemplo:** Algunos sistemas operativos no implementan estrategias explícitas para evitar o resolver interbloqueos, confiando en que los desarrolladores diseñen aplicaciones correctamente.

Sincronización de hilos

Monitores

- Un **monitor** es un mecanismo que combina sincronización y comunicación entre hilos.
- **Propósito:** Coordinar el acceso a recursos compartidos para evitar inconsistencias y garantizar el orden correcto de ejecución.
- **Conceptos Clave:**
  - Cada objeto en Java tiene un **monitor implícito**.
  - Métodos clave: *synchronized*, *wait*(), *notify*(), y *notifyAll*().
- **Ejemplo de la vida real:** Un semáforo en una intersección:
  - Solo un coche puede cruzar a la vez (sincronización).
  - Los coches esperan su turno (espera).
  - El semáforo indica a los coches cuándo avanzar (notificación).

Sincronización de hilos

Monitores

- **wait()**
  - Suspende el hilo actual hasta que otro hilo lo notifique.
  - Debe usarse dentro de un bloque *synchronized*.
- **notify()**
  - Despierta un hilo en espera.
- **notifyAll()**
  - Despierta a todos los hilos en espera en el monitor.

Nota:

- Todos estos métodos se usan con bloques *synchronized*.
- Los hilos deben "poseer el bloqueo del monitor" para usarlos.

Sincronización de hilos

Monitores

¿Cuándo usar monitores?

- Coordinación de hilos:
  - Ejemplo: Productores y consumidores.
- Acceso ordenado a recursos compartidos:
  - Evitar condiciones de carrera.
- Sincronización avanzada:
  - Cuando las soluciones básicas como synchronized no son suficientes.

Productores/Consumidores

- **Productor/Consumidor** es un modelo de comunicación entre hilos donde un hilo produce valores y otro hilo donde consume esos valores. Todo esto de manera sincronizada.

- Ambos hilos tratan de manejar valores de una zona común que se suele llamar **Monitor** (también se le suele llamar **Buffer**) que se encarga de sincronizar la entrada y salida de valores.

- Mediante **wait()** los hilos esperan a que el buffer tenga productos (en el caso de los consumidores) o capacidad para añadir nuevos productos (en el caso de los productores).

- Mediante **notifyAll()** los productores notifican que han añadido nuevos productos y los consumidores que han dejado hueco para nuevos productos al consumirlos.