

Clase 10 – Concurrencia

Programación concurrente.

Aplicaciones

Creación de un Threads

Ciclo de vida un Thread

Extendiendo de la clase Thread

Implementando la interfaz Runnable

Compartiendo objetos entre distintos Threads

Sincronización de Threads

Métodos synchronized

Métodos estáticos sincronizados

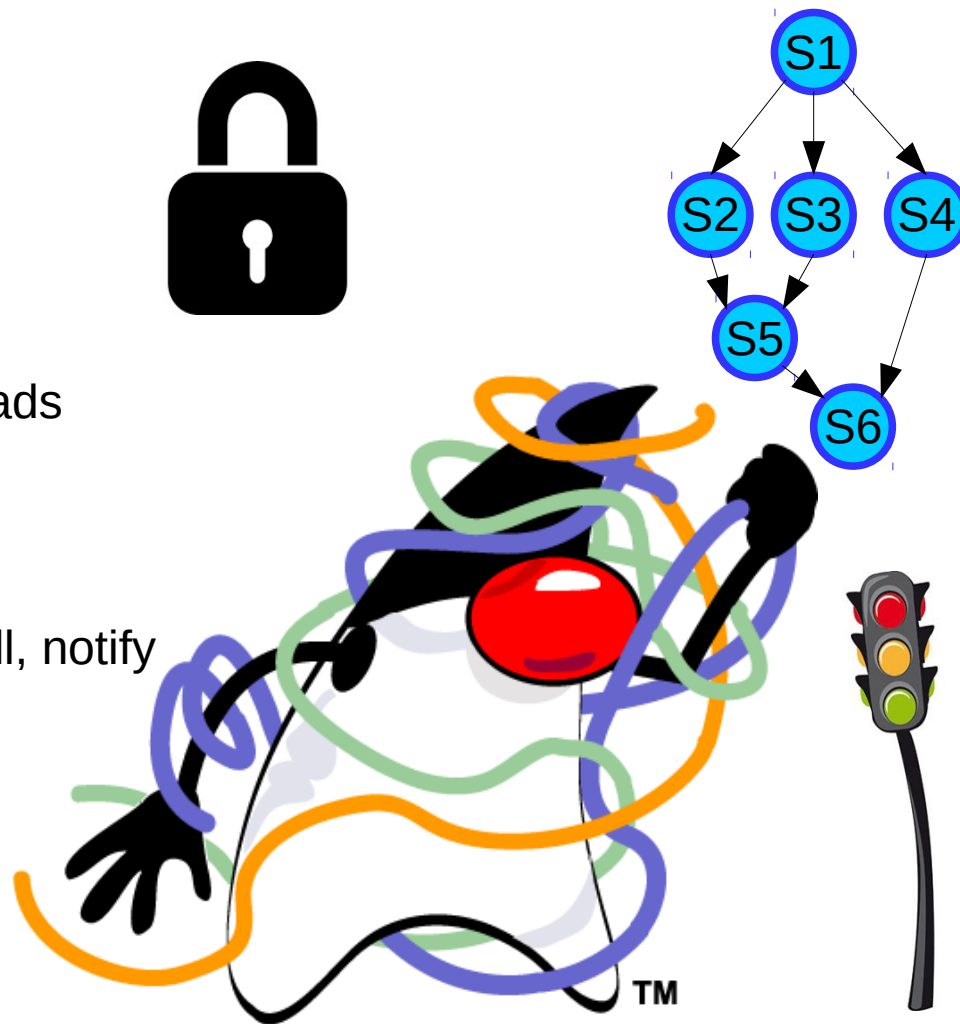
Comunicación entre Threads: wait, notifyAll, notify

La sentencia synchronized.

Bloqueo selectivo

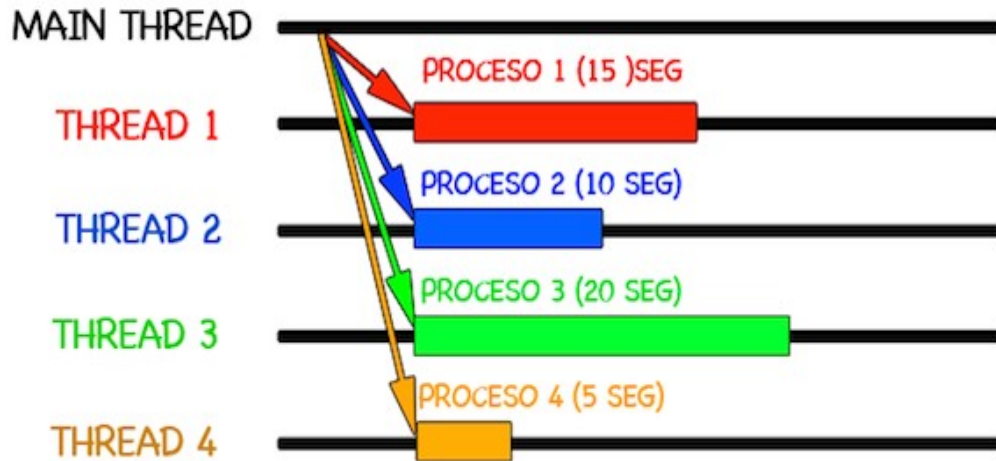
Semáforos

Métodos acquire y release



Clase 10 - Concurrencia

EL PROGRAMA TARDE EN EJECUTARSE 20 SEGUNDOS
QUE ES EL TIEMPO DEL PROCESO MÁS LARGO



Una gran cantidad de problemas de programación pueden resolverse utilizando programación secuencial. Sin embargo, para algunos problemas, resulta conveniente e incluso esencial ejecutar varias partes del programa en paralelo, de modo que dichas partes parezcan estarse ejecutando concurrentemente o, si hay disponibles varios procesadores, se ejecuten realmente de manera simultánea.

Programación Concurrente

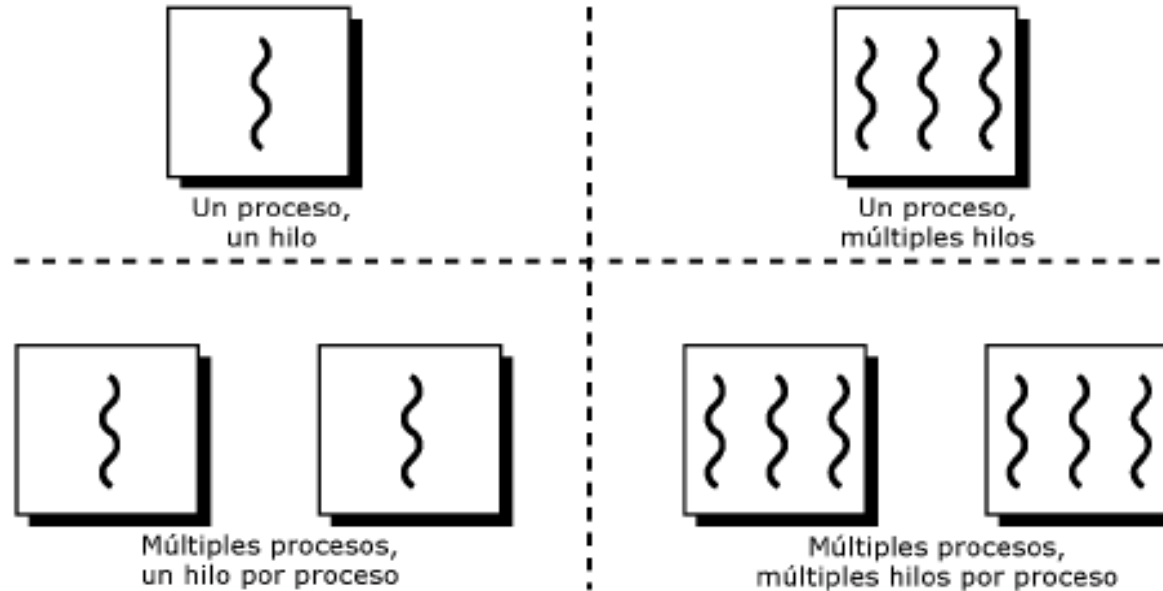
Se conoce por programación concurrente a la rama de la informática que trata de las técnicas de programación que se usan para expresar el **paralelismo** entre tareas y para resolver los problemas de **comunicación y sincronización** entre procesos.

El principal problema de la programación concurrente corresponde a **no saber en que orden** se ejecutan los programas (en especial los programas que se comunican). Se debe tener especial **cuidado** en que este orden no afecte el resultado de los programas.

Programación Concurrente

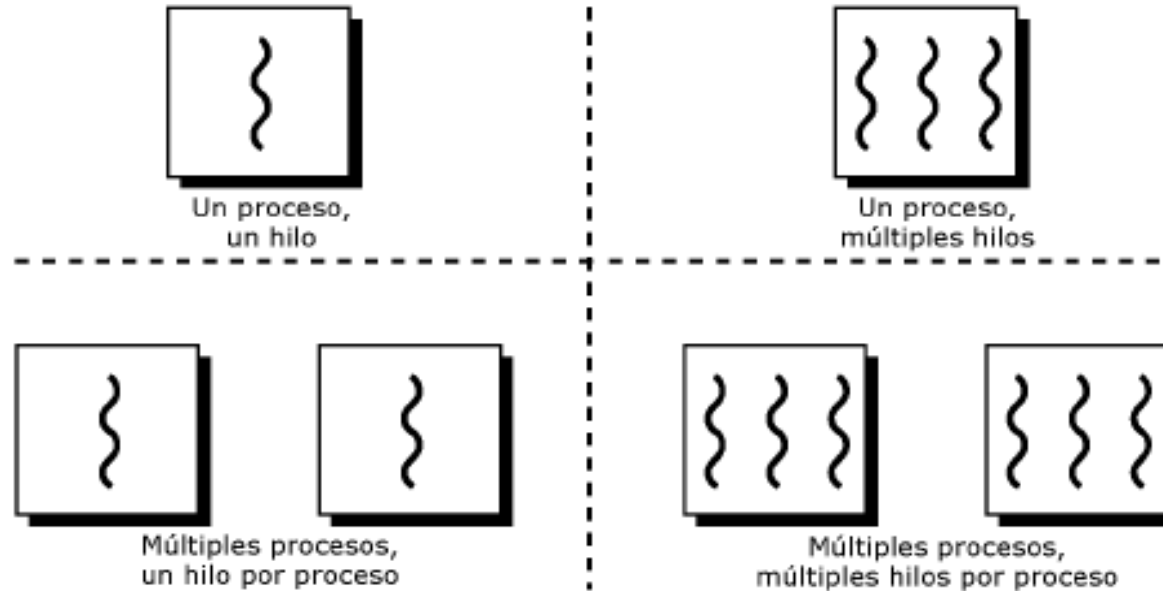
- Son programas que tienen **múltiples líneas de flujo de control**.
- Las sentencias de un programas concurrente se ejecutan de acuerdo con un **orden no estricto**.
- La secuencialización de un programa concurrente es entre hitos o puntos de sincronización.
- Un programa concurrente se suele concebir como un conjunto de procesos que **colaboran y compiten entre sí**.

Programación Concurrente



Una forma muy sencilla de implementar la concurrencia es en el nivel del sistema operativo, utilizando *procesos*. Un proceso es un programa autocontenido que se ejecuta en su propio espacio de direcciones. Un S.O. *multitarea* puede ejecutar más de un proceso (programa) simultáneamente, conmutando periódicamente la CPU de un proceso a otro, lo que aparenta que cada proceso se ejecuta por separado y al mismo tiempo.

Programación Concurrente



Los sistemas concurrentes, como los que utiliza Java, comparten recursos tales como la memoria, E/S, por lo que la dificultad está en coordinar el uso de estos recursos entre las distintas tareas dirigidas por hilos, de modo que no haya más de una tarea en cada momento que pueda acceder a un determinado recurso.

Programación Concurrente

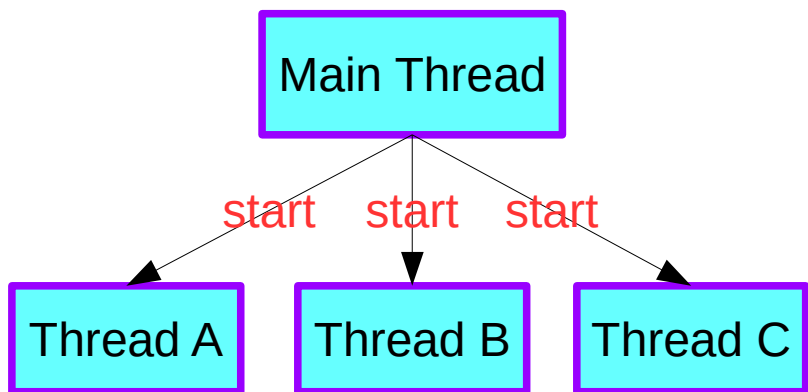
Aplicaciones clásicas:

- Programación de sistemas multicomputadores.
- Sistemas operativos.
- Control y monitorización de sistemas físicos reales.

Aplicaciones actuales:

- Servicios WEB.
- Sistemas multimedia.
- Cálculo numérico.
- Procesamientos entrada/salida.
- Simulación de sistemas dinámicos.
- Interacción operador/máquina (GUIs)
- Tecnologías de componentes.
- Código móvil.
- Sistemas embebidos.

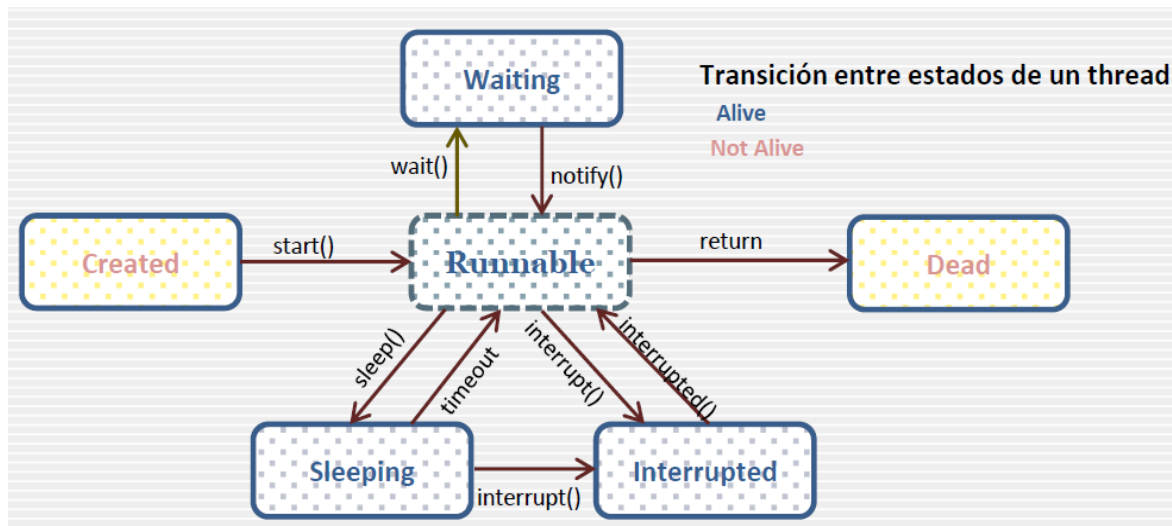
Hilos (Threads) en Java



La clase Thread dispone de una serie de métodos para caracterizar el thread/hilo en el programa: `isAlive()`, `isDaemon()`, `setName()`, `setDaemon()`, `run()`, etc.

Java proporciona un API para el uso de hilos: clase Thread dentro del paquete `java.lang.Thread`.

Cuando arranca un programa existe un hilo principal (main), y luego se pueden generar nuevos hilos que ejecutan código en objetos diferentes o el mismo.



Creación de un Thread

Para crear un Thread (hilo de control), se crea un objeto de tipo Thread:

```
Thread trabajador = new Thread();
```

Tras crearlo, se puede configurar y ejecutar.

Para ejecutarlo se invoca el método **start()**.

El método **start()** lanza un nuevo hilo de control basado en los datos del objeto Thread, y luego retorna.

```
trabajador.start();
```

Al lanzarse el método **start()**, la máquina virtual invoca al nuevo método **run()** del hilo de control, con lo que el hilo se activa.

```
public class HiloBasico extends Thread
{
    int cont;
```

```
HiloBasico(int c)
{
    cont = c;
}
```

```
public void run()
{
    while (true)
    {
        System.out.println(cont);
    }
}
```

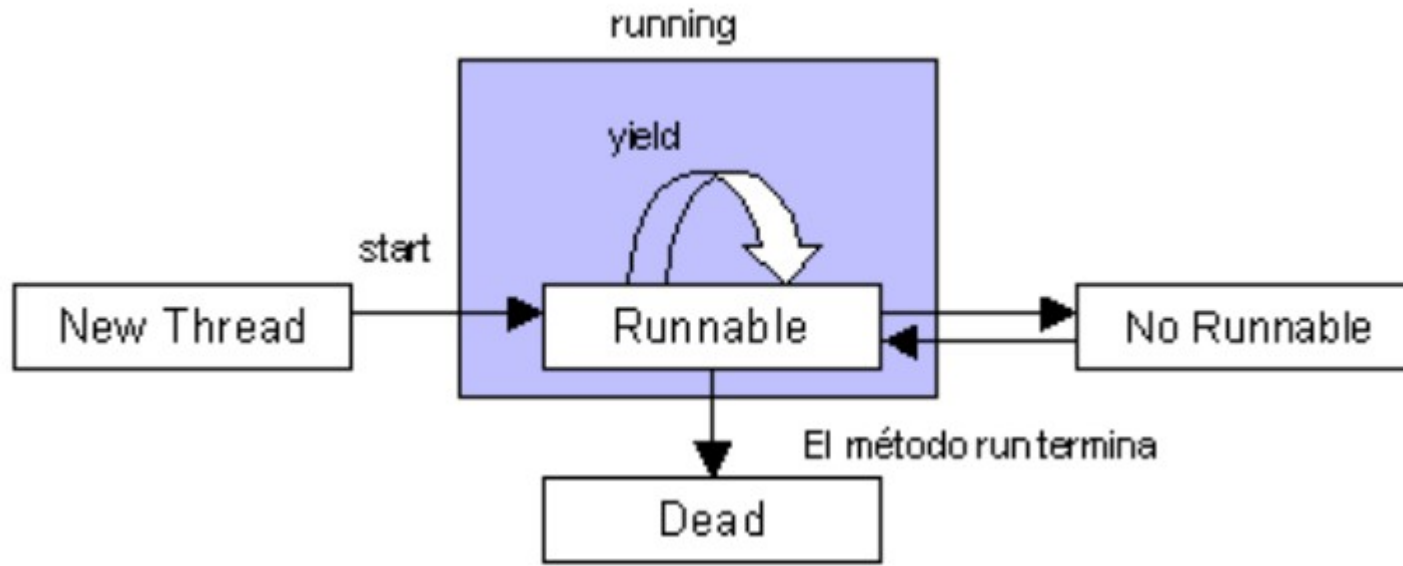
```
public static void main(String[] args)
{
    new HiloBasico(0).start();
    new HiloBasico(1).start();
}
```

Creación de un Thread

Sólo se puede invocar **start()** una vez para cada hilo (de invocarlo nuevamente se lanza la excepción `IllegalThreadStateException`).

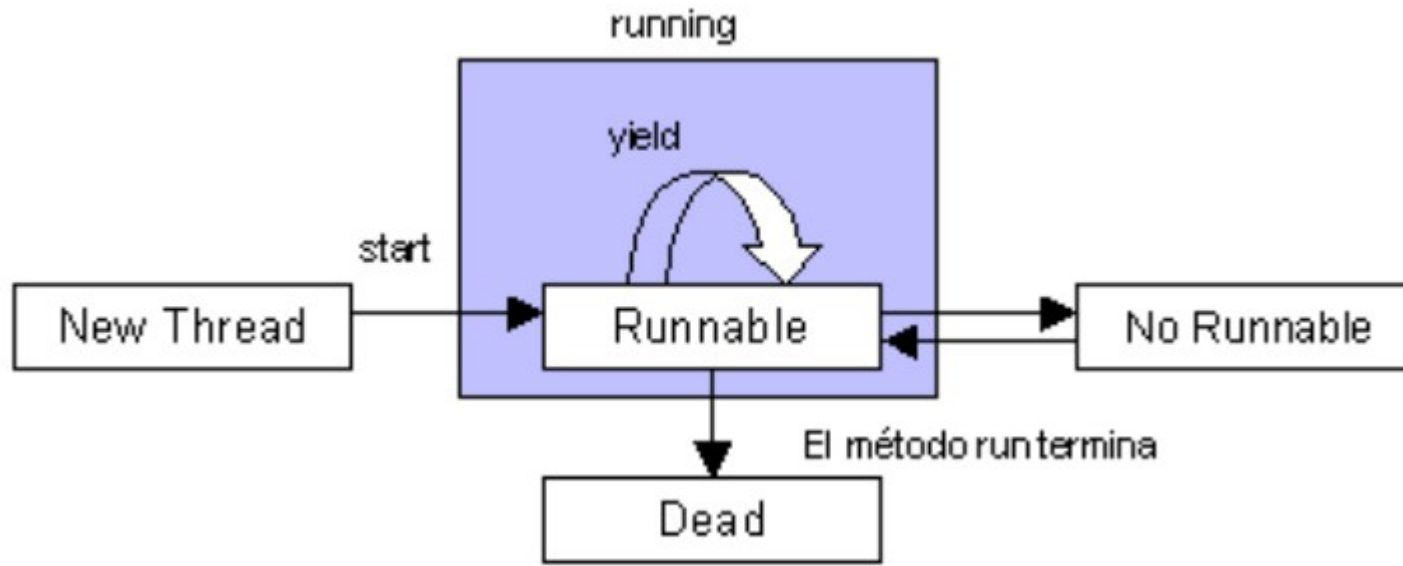
Cuando se retorna de un método **run()** de un hilo, el hilo ha finalizado.

Ciclo de Vida de un Thread



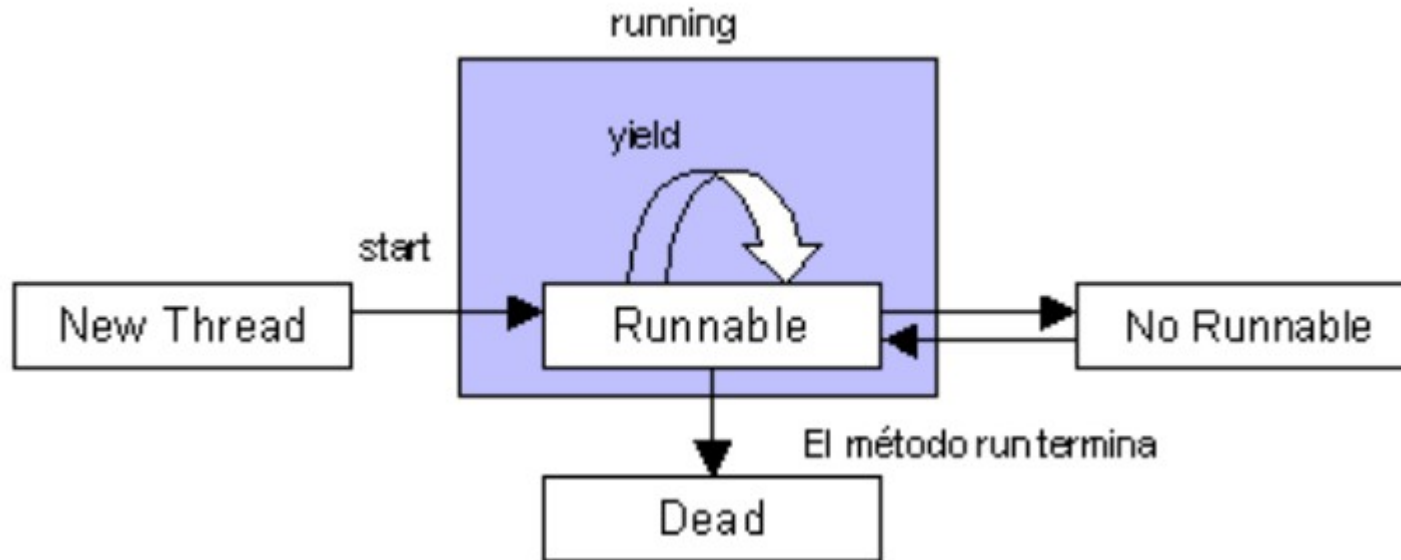
Cuando se instancia la clase Thread (o una subclase) se crea un nuevo Thread que está en su estado inicial ('New Thread' en el gráfico). En este estado es simplemente un objeto más. No existe todavía el thread en ejecución.

Ciclo de Vida de un Thread



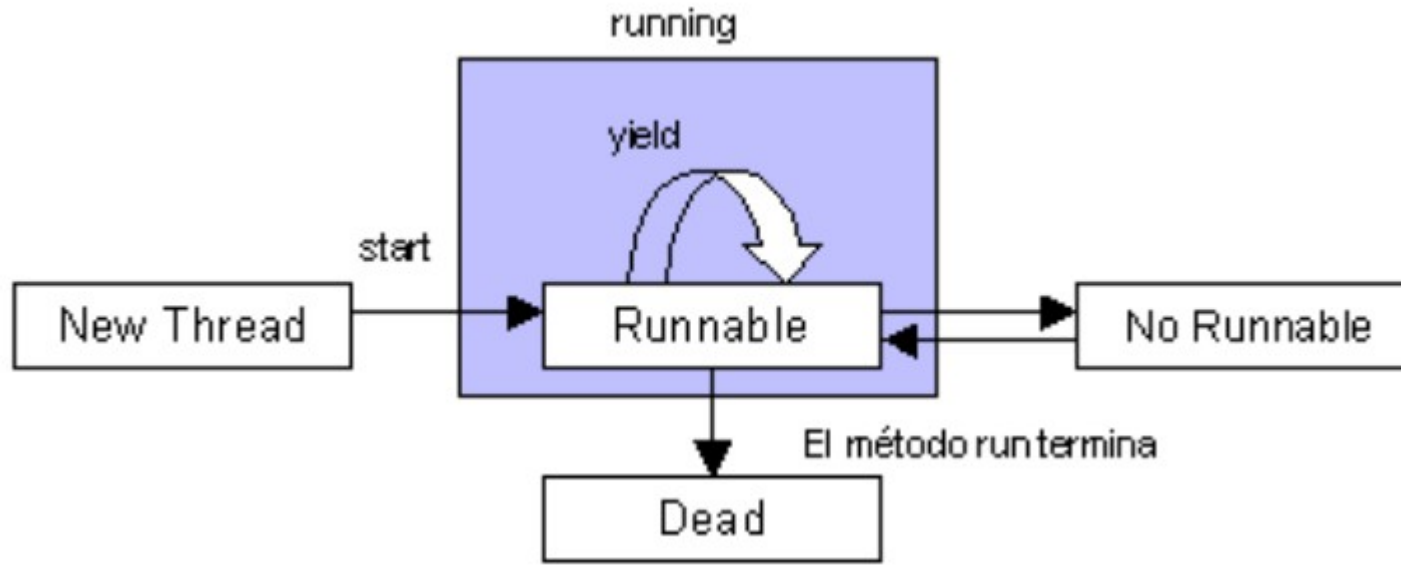
El único método que puede invocarse sobre él es el método `start()`. Cuando se invoca el método `start()` sobre el hilo, el sistema crea los recursos necesarios, lo planifica (le asigna prioridad) y llama al método `run()`. En este momento el hilo está corriendo, se encuentra en el estado 'runnable'.

Ciclo de Vida de un Thread



Si el método `run()` invoca internamente el método `sleep()` o `wait()` o el hilo tiene que esperar por una operación de entrada/salida, entonces el hilo pasa al estado 'no runnable' (no ejecutable) hasta que la condición de espera finalice. Durante este tiempo el sistema puede ceder control a otros hilos activos.

Ciclo de Vida de un Thread



Por último cuando el método run finaliza el hilo termina y pasa a la situación 'Dead' (Muerto).

Ciclo de Vida de un Thread

La finalización de un hilo suele ocurrir cuando se termina de ejecutar el método `run()`.

No obstante, hay otras tres razones por las que un hilo puede terminar:

- En algún momento de la ejecución del método `run()` se ha generado una **excepción** que nadie ha capturado. La excepción se propaga hasta el propio método `run()`. Si tampoco éste tiene un manejador para la excepción, el método `run()` finaliza abruptamente, terminando la ejecución de la hilo.
- Si se llama al método **`stop()`** del hilo.
- Cuando se llama al método **`destroy()`** del hilo.

En cualquiera de los casos, el hilo finaliza su ejecución, y deja de ser tenida en cuenta por el planificador.

Creación de un Thread

La implementación estándar de **Thread.run()**, no hace nada.

Para obtener un hilo que haga algo, se debe hacer una de estas dos posibilidades:

Thread



Mi_Thread

1) Crear una clase que extienda de Thread y sobrescribir el método run.

2) Crear una clase que implemente la interfaz Runnable y crear un objeto de tipo Thread a partir de éste.

Runnable



Mi_Runnable

```
Thread unThread = new Thread ( new Mi_Runnable());
```


Extendiendo de la clase Thread

Thread



HiloBasico

- 1) Se crea una clase que extienda de Thread
- 2) Se implementa el método run()
- 3) Luego de instanciar la clase creada, se invoca el método start(), que a su vez invoca el método run()

```
public class HiloBasico extends Thread
{
    int cont;

    HiloBasico(int c)
    {
        cont = c;
    }

    public void run()
    {
        while (true)
        {
            System.out.println(cont);
        }
    }

    public static void main(String[] args)
    {
        new HiloBasico(0).start();
        new HiloBasico(1).start();
    }
}
```

Extendiendo de la clase **Thread**

La ejecución de la aplicación causa (en teoría) la salida por pantalla de líneas con 0's y 1's entrelazados. Eso se debe a que la máquina virtual distribuye el tiempo del procesador entre los dos Hilos, de modo que según quién tenga el procesador se escribirá un 0 o un 1.

Este modo de implementar nuevos hilos tiene una desventaja, debido a la herencia simple.

```
public class HiloBasico extends Thread
{
    int cont;

    HiloBasico(int c)
    {
        cont = c;
    }

    public void run()
    {
        while (true)
        {
            System.out.println(cont);
        }
    }

    public static void main(String[] args)
    {
        new HiloBasico(0).start();
        new HiloBasico(1).start();
    }
}
```

Implementando Runnable

Runnable



RunnableBasico

```
public class RunnableBasico implements Runnable
{
    int cont;

    RunnableBasico(int c)
    {
        cont = c;
    }

    public void run()
    {
        while (true)
        {
            System.out.println(cont);
        }
    }

    public static void main(String[] args)
    {
        new Thread(new RunnableBasico(0)).start();
        new Thread(new RunnableBasico(1)).start();
    }
}
```

```
Thread unThread = new Thread ( new RunnableBasico());
```

Implementando Runnable

Es posible comenzar la ejecución del método `run()` de cualquier objeto que implemente el interfaz en un hilo independiente. Para eso, es suficiente crear un nuevo objeto de la clase `Thread`, pasándole en el constructor el objeto que se desea ejecutar.

Cuando se llame al método `start()` de esa hebra, el método que se ejecutará será, realmente, el método `run()` del objeto pasado en el constructor.

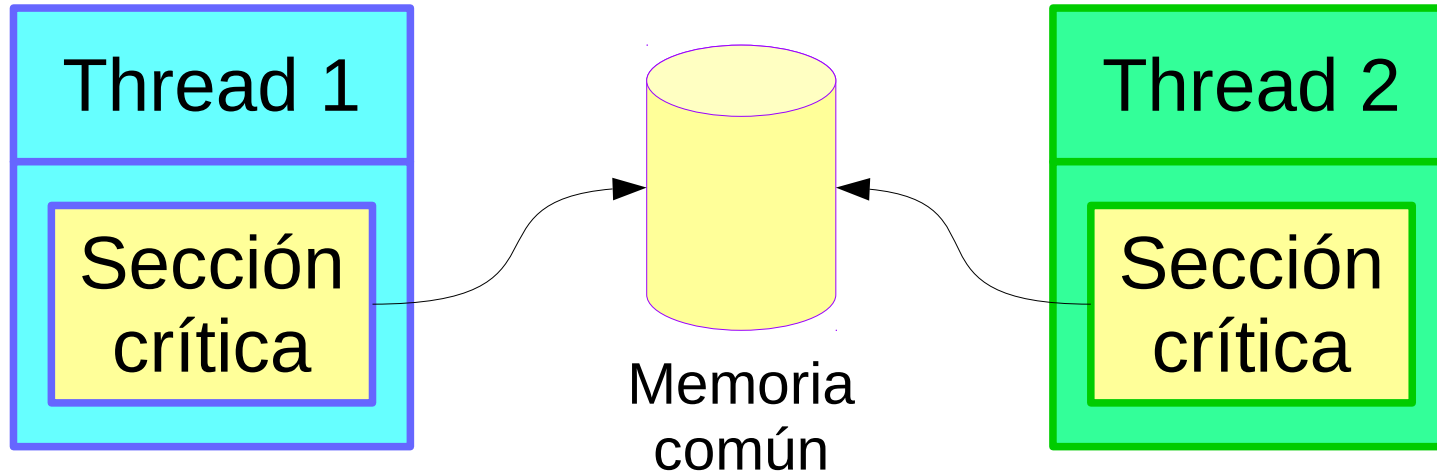
```
public class RunnableBasico implements Runnable
{
    int cont;

    RunnableBasico(int c)
    {
        cont = c;
    }

    public void run()
    {
        while (true)
        {
            System.out.println(cont);
        }
    }

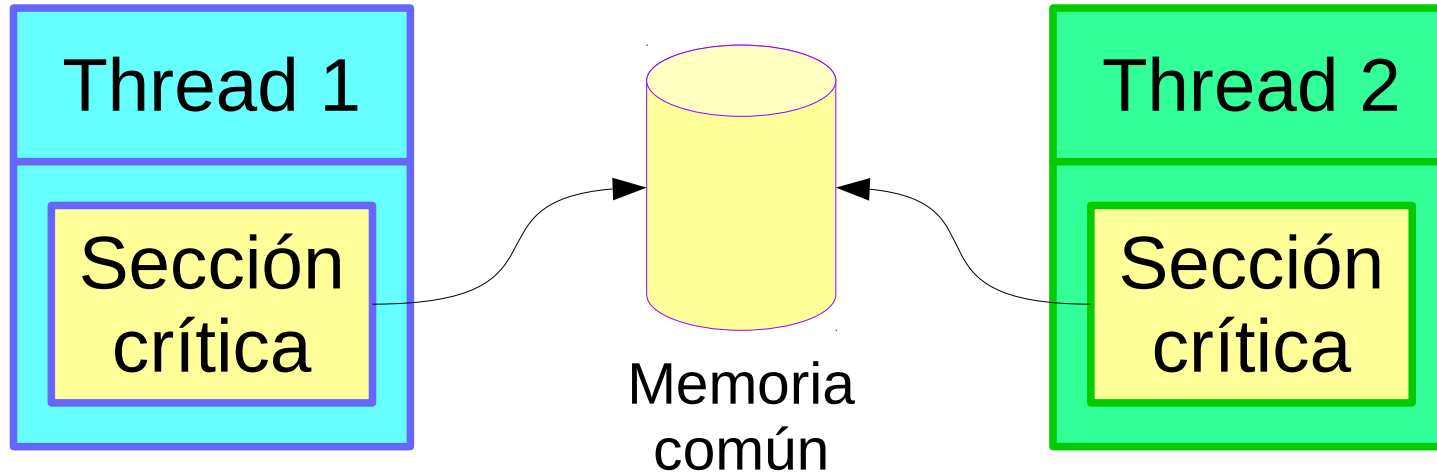
    public static void main(String[] args)
    {
        new Thread(new RunnableBasico(0)).start();
        new Thread(new RunnableBasico(1)).start();
    }
}
```

Compartiendo objetos entre distintos Threads



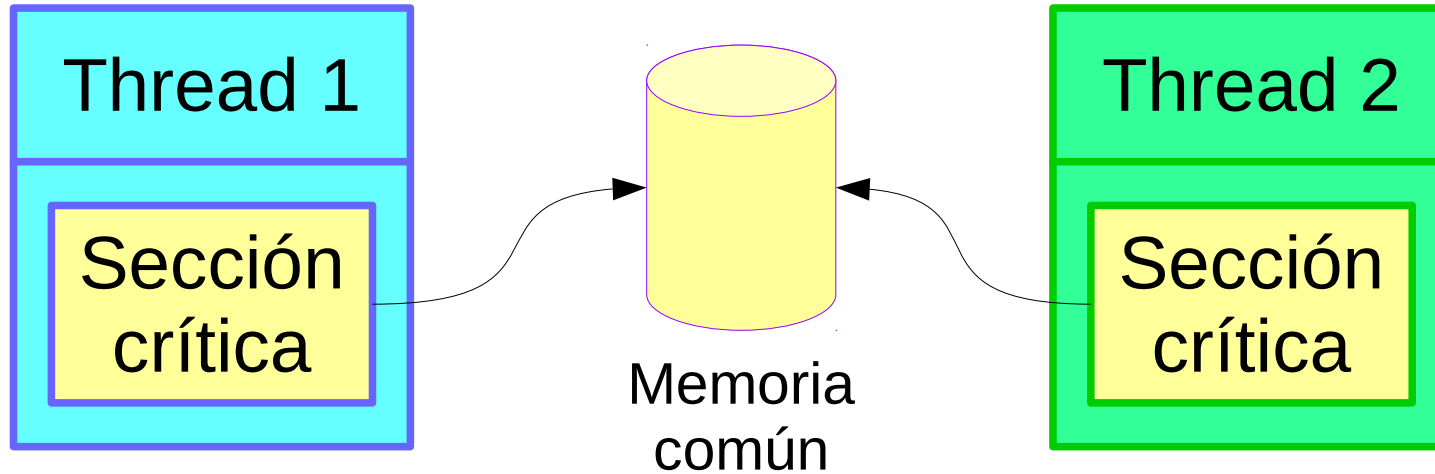
Cuando varios Threads *comparten* un mismo objeto y modifican su estado, pueden ocurrir inconsistencias en la visión que tenga cada thread del objeto.

Compartiendo objetos entre distintos Threads



Al estar en un ambiente concurrente, si no se establece una **sincronización** en el acceso al objeto compartido (de a un thread por vez), los resultados son imprevisibles.

Compartiendo objetos entre distintos Threads



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.

Ejemplo

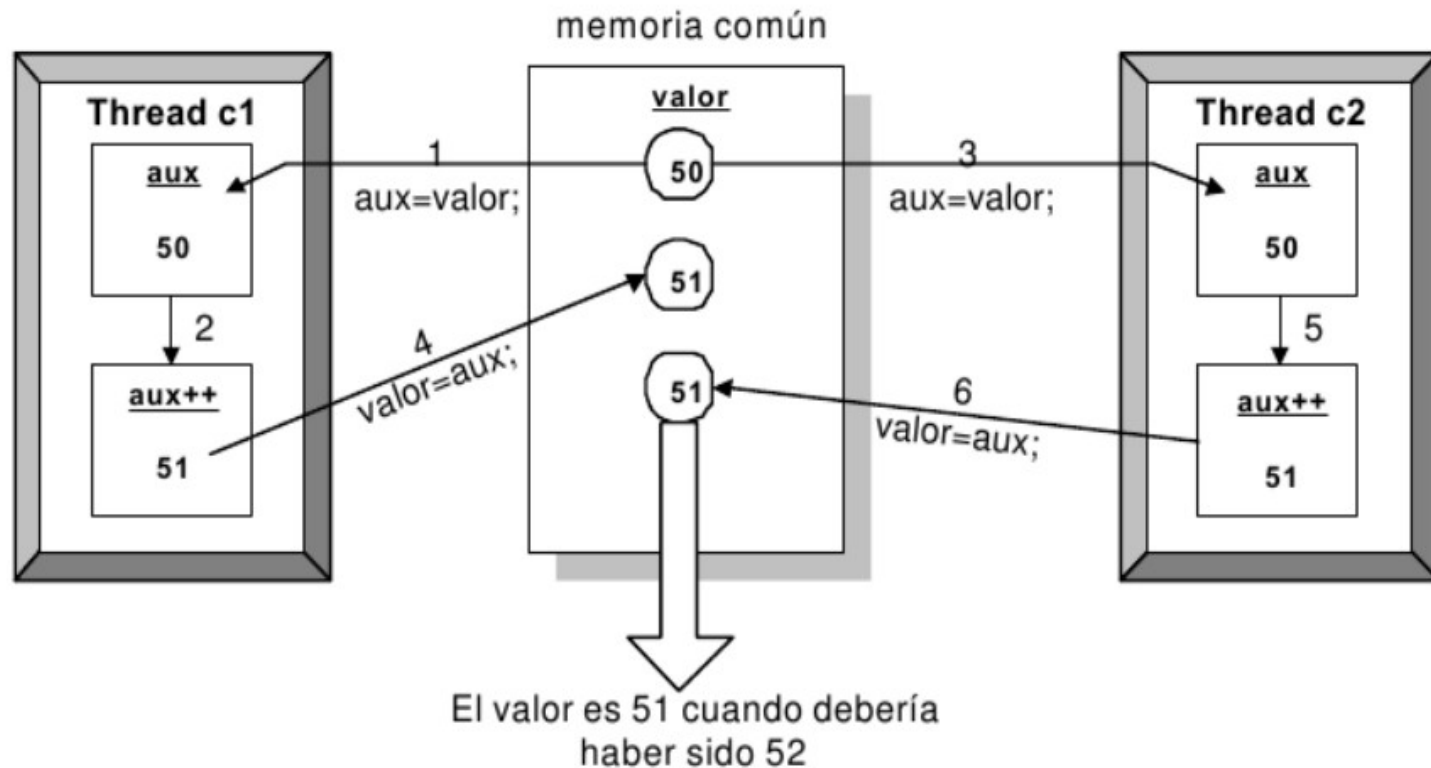
```
class ThreadSinSync
{
    public static void main(String arg[])
    {
        Contable c1, c2;
        Contador c = new Contador();
        c1 = new Contable(c);
        c2 = new Contable(c);
        c1.start();
        c2.start();
    }
}
```

La salida por pantalla es:
Contado hasta ahora:
124739
Contado hasta ahora:
158049
cuando debería haber
sido: en la primera línea
un número comprendido
entre 100000 y 200000;
y en la segunda línea el
valor 200000.

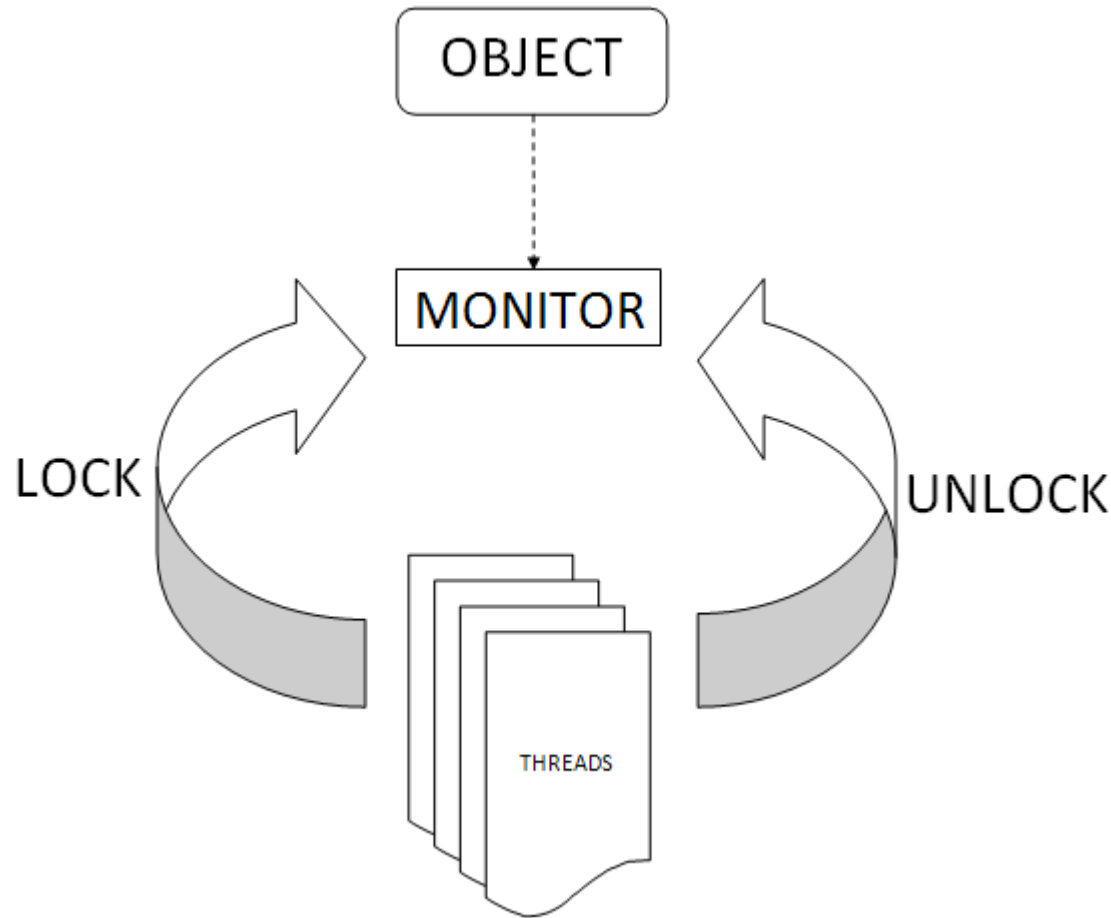
Por qué sucedió eso?

```
class Contador
{
    private long valor = 0;
    public void incrementa()
    {
        long aux;
        aux = valor;
        aux++;
        valor = aux;
    }
    public long getValor()
    {
        return valor;
    }
}
```

	valor	aux
1	50	50
2	50	51
3	50	50
4	50	50
5	50	51
6	51	51

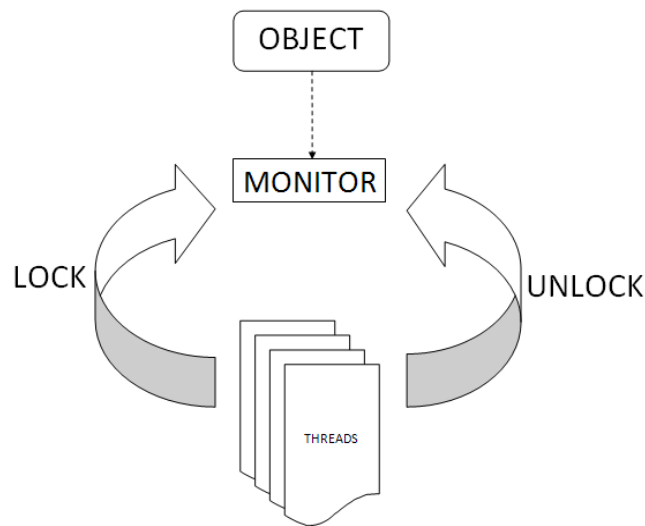


Monitores



En la programación paralela, los monitores son **objetos destinados a ser usados sin peligro por más de un hilo de ejecución**. La característica que principalmente los define es que sus **métodos son ejecutados con exclusión mutua**. Lo que significa, que en cada momento en el tiempo, un hilo como máximo puede estar ejecutando cualquiera de sus métodos.

Monitores - Componentes



Un monitor tiene cuatro componentes: inicialización, datos privados, métodos del monitor y cola de entrada.

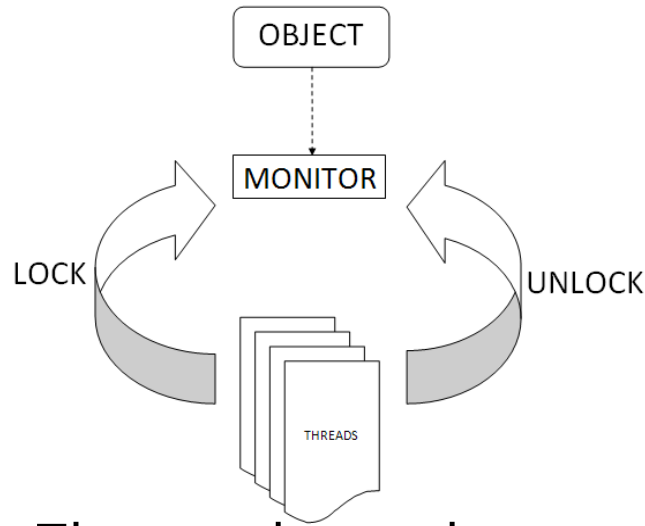
Inicialización: contiene el código a ser ejecutado cuando el monitor es creado

Datos privados: contiene los procedimientos privados, que sólo pueden ser usados desde dentro del monitor y no son visibles desde fuera

Métodos del monitor: son los procedimientos que pueden ser llamados desde fuera del monitor.

Cola de entrada: contiene a los hilos que han llamado a algún método del monitor pero no han podido adquirir permiso para ejecutarlos aún.

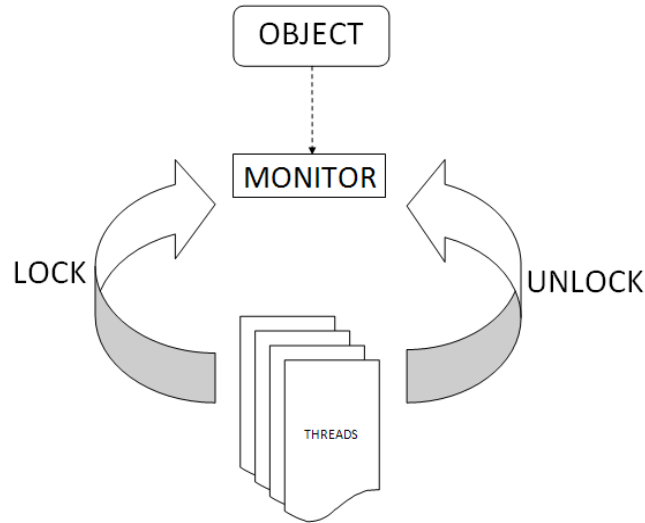
Monitores – Exclusión mutua



Los monitores están pensados para ser usados en **entornos multiproceso** o multihilo, y por lo tanto muchos procesos o threads pueden llamar a la vez a un procedimiento del monitor. Los monitores garantizan que en cualquier momento, **a lo sumo un thread** puede estar ejecutando dentro de un monitor.

Ejecutar dentro de un monitor significa que sólo un thread estará en estado de ejecución mientras dura la llamada a un procedimiento del monitor. Para **garantizar la integridad de los datos privados**, el monitor hace cumplir la exclusión mutua implícitamente, de modo que sólo un procedimiento esté siendo ejecutado a la vez. De esta forma, si **un thread** llama a un procedimiento mientras otro thread está dentro del monitor, **se bloqueará y esperará** en la cola de entrada hasta que el monitor quede nuevamente libre.

Monitores – Exclusión mutua



Para que resulten útiles en un entorno de concurrencia, los monitores deben incluir algún tipo de forma de **sincronización**. Por ejemplo, supóngase un thread que está dentro del monitor y necesita que se cumpla una condición para poder continuar la ejecución. En ese caso, se debe contar con un mecanismo de **bloqueo del thread**, a la vez que se debe **liberar el monitor para ser usado por otro hilo**. Más tarde, cuando la condición permita al thread bloqueado continuar ejecutando, debe poder ingresar en el monitor en el mismo lugar donde fue suspendido. Para esto los monitores poseen variables de condición que son accesibles sólo desde adentro.

Estructura sintáctica de un Monitor

```
class Mi_Monitor
{
    //definir aquí datos protegidos por el monitor

    public Mi_Monitor()
    {...} //constructor

    public synchronized tipo1 metodo1() throws InterruptedException
    {
        while(!condicion1)
            wait();

        ...
        notifyAll();
    }

    public synchronized tipo2 metodo2() throws InterruptedException
    {...}
```

Estructura sintáctica de un Monitor

Analicemos cada componente por parte y en forma progresiva

Sincronización de Threads



Para evitar corromper la zona crítica de un recurso compartido (objeto referido por varios threads en forma concurrente) es necesario que los mensajes enviados al recurso por parte de cada thread se realicen de a uno por vez.

A esta acción sincronizada se la denomina ***bloqueo de un objeto.***

Para indicar esta acción se utiliza la sentencia
synchronized

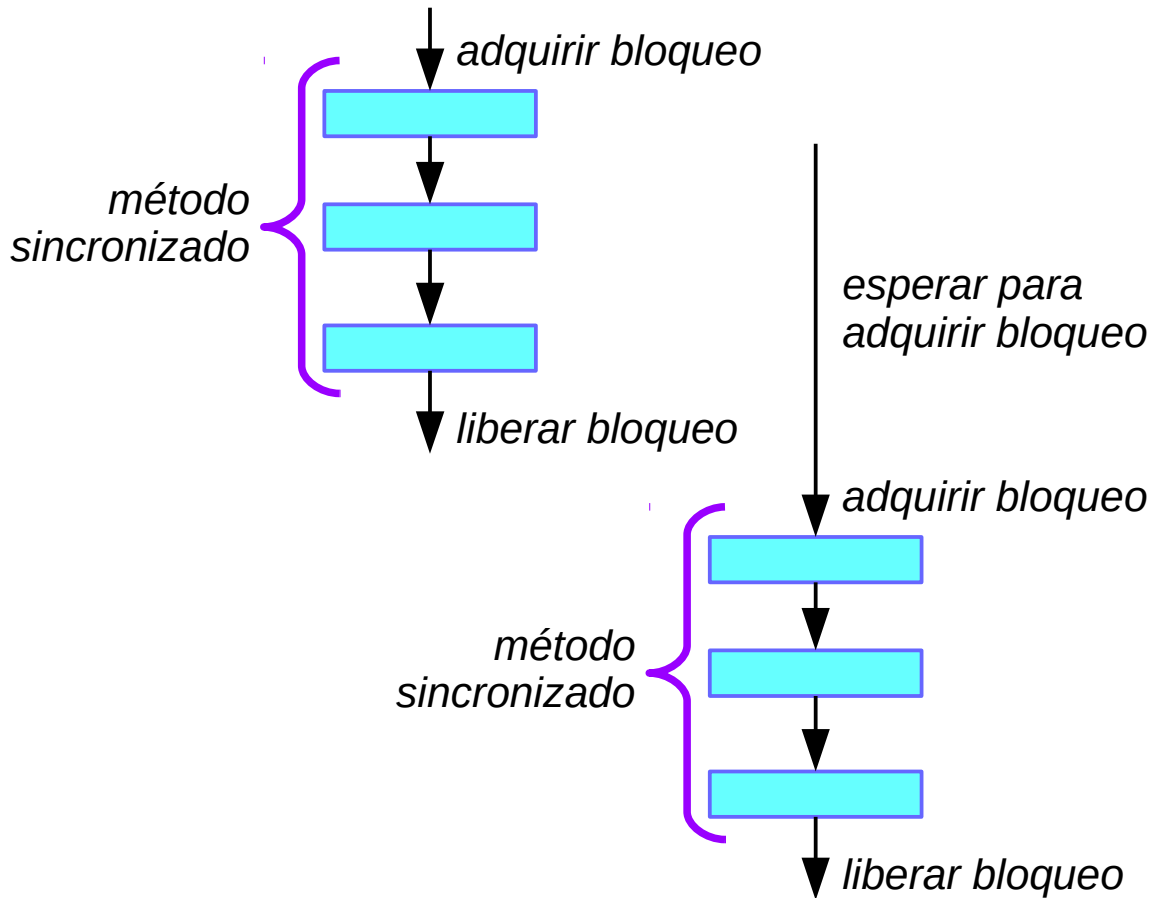
Sincronización de Threads



Cada thread debe ***adquirir el bloqueo de un objeto*** antes de enviarle un mensaje que pueda corromper la zona crítica.

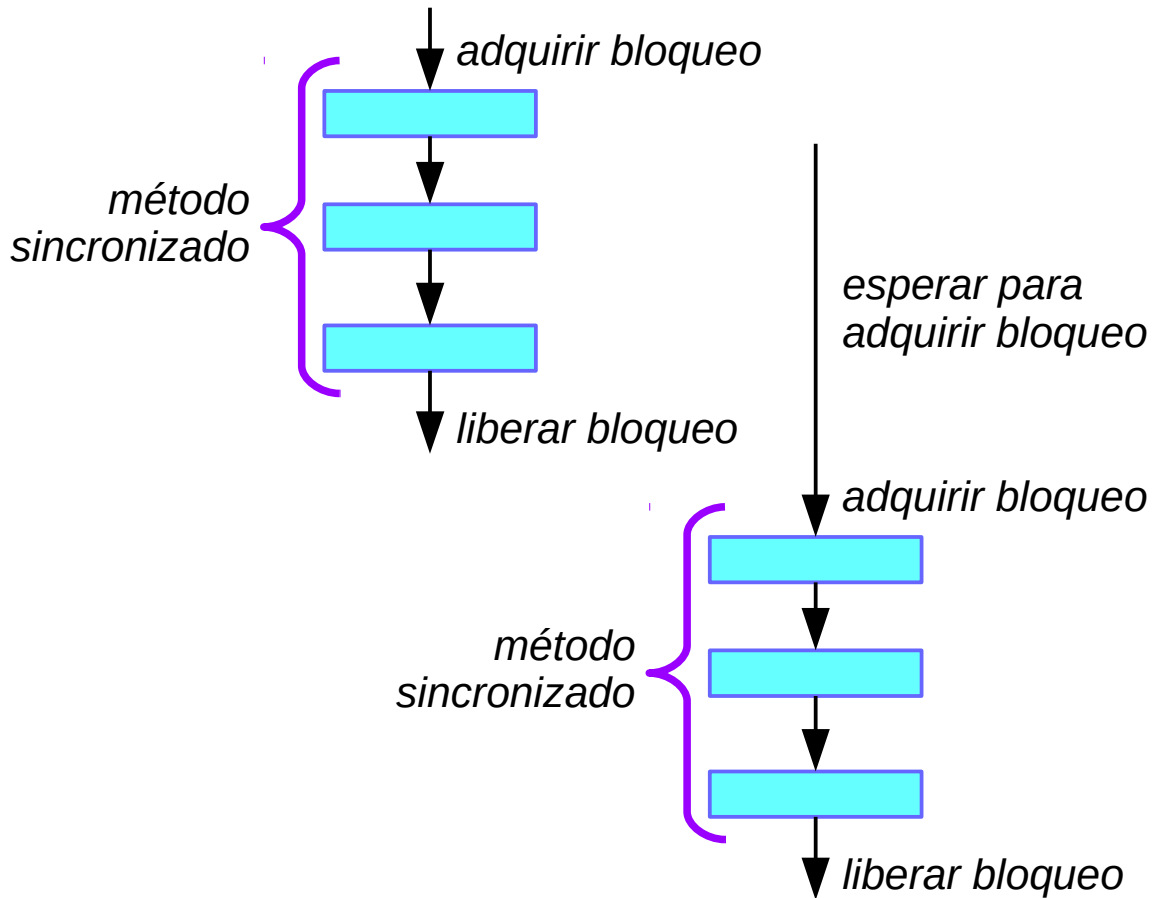
Adquirir el bloqueo de un objeto impide que cualquier otro thread lo adquiera, hasta que el propietario lo libere.

Métodos **synchronized**



Si un hilo invoca a un método **synchronized** sobre un objeto (recurso compartido), en primer lugar se adquiere el bloqueo de ese objeto, se ejecuta el cuerpo del método y después se libera el bloqueo.

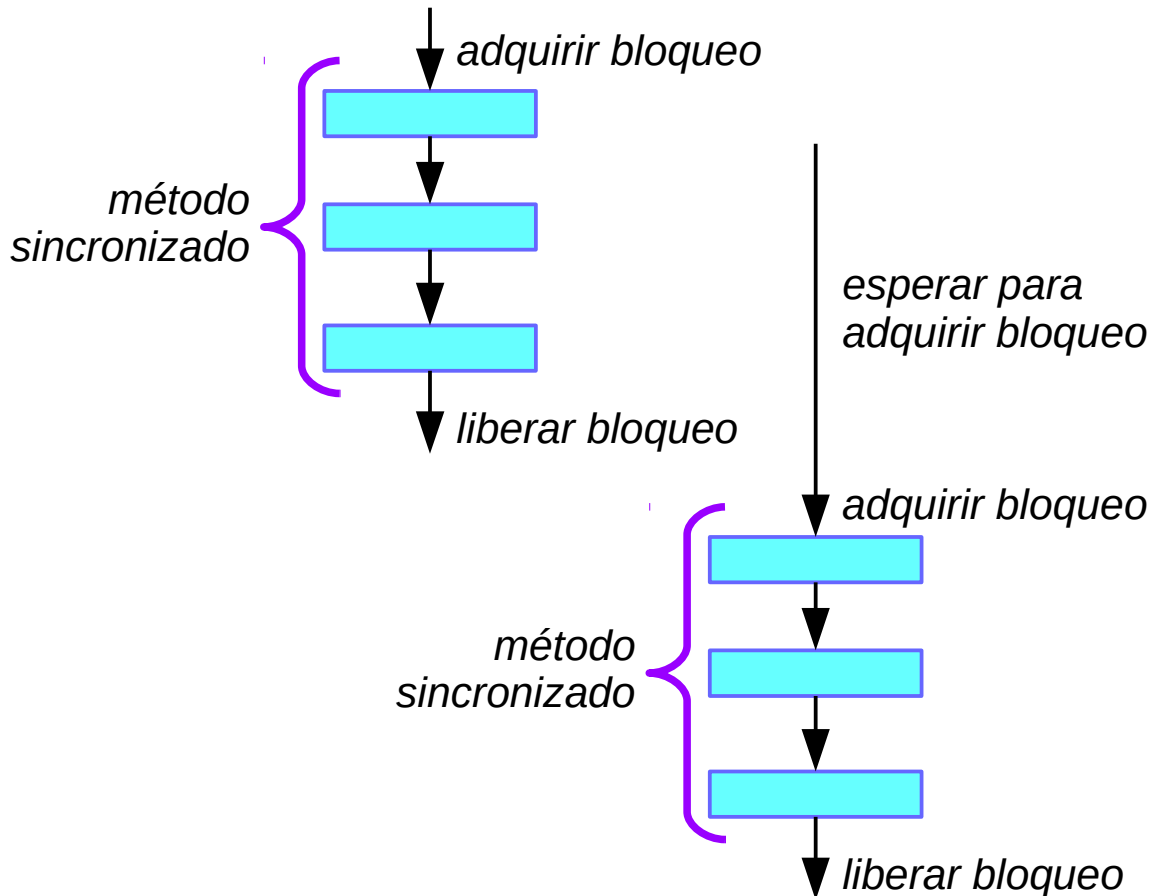
Métodos **synchronized**



Otro hilo que invoque un método **synchronized** sobre ese mismo objeto se bloqueará hasta que el hilo anterior libere el bloqueo. O sea, debe esperar hasta que el recurso esté disponible.

Si varios hilos esperan por un recurso, no se sabe cuál será el siguiente en obtenerlo.

Métodos **synchronized**



Ejecución mutuamente exclusiva en el tiempo.

Los accesos NO sincronizados no esperan por ningún bloqueo.

La posesión de los bloqueos es por hilo. Al invocar a un método sincronizado desde el interior de otro método sincronizado sobre el mismo objeto, se procederá sin bloqueo. Se libera el bloqueo cuando el método sincronizado más externo retorne.

Métodos **synchronized**

Sintaxis

```
public synchronized tipo metodo()  
{  
    . . .  
}
```

Métodos **synchronize**: implicancias

Los constructores no necesitan ser sincronizados (de hecho no pueden declararse sincronizados)

El acceso a los campos debe hacerse a través métodos públicos exclusivamente, ya que es la única manera de evitar corromper la zona crítica

No hay garantías del orden en que se ejecutarán los métodos sincronizados al competir por el recurso

Métodos **synchronize**: implicancias

Cuando una clase extendida redefine a un método sincronizado (en la clase base), el nuevo método puede ser sincronizado o no.

Los requerimientos de sincronización son parte de la implementación de una clase.

Métodos estáticos sincronizados

Los métodos estáticos también se pueden declarar sincronizados.

Dos hilos no pueden ejecutar métodos estáticos sincronizados de la misma clase al mismo tiempo.

La adquisición del bloqueo en un método estático sincronizado no afecta a los objetos de esa clase. Solo se bloquean otros métodos estáticos sincronizados.

Métodos **synchronized**: Ejemplo *productor - consumidor*

```
public class Contenedor
{
    private int valor;

    public Contenedor()
    {
        super();
    }

    public synchronized void put(int nv)
    {
        valor = nv;
        System.out.println("metido el valor: " + valor);
    }

    public synchronized int get()
    {
        System.out.println("sacado el valor: " + valor);
        //valor = 0;
        return valor;
    }
}
```

Métodos **synchronized**: Ejemplo *productor - consumidor*

```
public class Productor extends Thread
{
    private Contenedor contenedor;

    public Productor(Contenedor c)
    {
        contenedor = c;
    }

    public void run()
    {
        int i;
        for (i = 0; i < 10; i++)
        {
            contenedor.put(i);
            try
            {
                sleep((int) (Math.random() * 100));
            }
            catch (InterruptedException e)
            {
            }
        }
    }
}
```

```
public class Consumidor extends Thread
{
    private Contenedor contenedor;

    public Consumidor(Contenedor c)
    {
        contenedor = c;
    }

    public void run()
    {
        int value = 0;
        int i;
        for (i = 0; i < 10; i++)
        {
            value = contenedor.get();
            try
            {
                sleep((int) (Math.random() * 100));
            }
            catch (InterruptedException e)
            {
            }
        }
    }
}
```

Métodos **synchronized**: Ejemplo *productor - consumidor*

```
public class Prueba
{
    public Prueba ()
    {
        super ();
    }

    public static void main(String args[])
    {
        Contenedor recursoCompartido = new Contenedor ();
        Productor productor = new Productor(recursoCompartido);
        Consumidor consumidor = new Consumidor(recursoCompartido);
        productor.start ();
        consumidor.start ();
    }
}
```

Métodos **synchronized**: Ejemplo *productor - consumidor*

Tanto el hilo productor como el consumidor comparten el mismo objeto de la clase Caja. Para asegurar la exclusión mutua en la zona crítica (la que accede a valor), se declaran los métodos `get()` y `put()` como `synchronized`.

Pero esto no es suficiente para que el programa funcione correctamente.

la salida por pantalla
podría ser la siguiente:

```
metido el valor: 0
sacado el valor: 0
sacado el valor: 0
metido el valor: 1
sacado el valor: 1
sacado el valor: 1
metido el valor: 2
metido el valor: 3
sacado el valor: 3
metido el valor: 4
sacado el valor: 4
metido el valor: 5
sacado el valor: 5
sacado el valor: 5
metido el valor: 6
metido el valor: 7
metido el valor: 8
sacado el valor: 8
sacado el valor: 8
metido el valor: 9
```

Métodos **synchronized**: Ejemplo *productor - consumidor*

De alguna forma habría que asegurar que no ocurran errores, tales como sacar un elemento de la caja cuando está vacía.

En nuestro ejemplo, el error se refleja en los renglones destacados

la salida por pantalla
podría ser la siguiente:

```
metido el valor: 0
sacado el valor: 0
sacado el valor: 0
metido el valor: 1
sacado el valor: 1
sacado el valor: 1
metido el valor: 2
metido el valor: 3
sacado el valor: 3
metido el valor: 4
sacado el valor: 4
metido el valor: 5
sacado el valor: 5
sacado el valor: 5
metido el valor: 6
metido el valor: 7
metido el valor: 8
sacado el valor: 8
sacado el valor: 8
metido el valor: 9
```

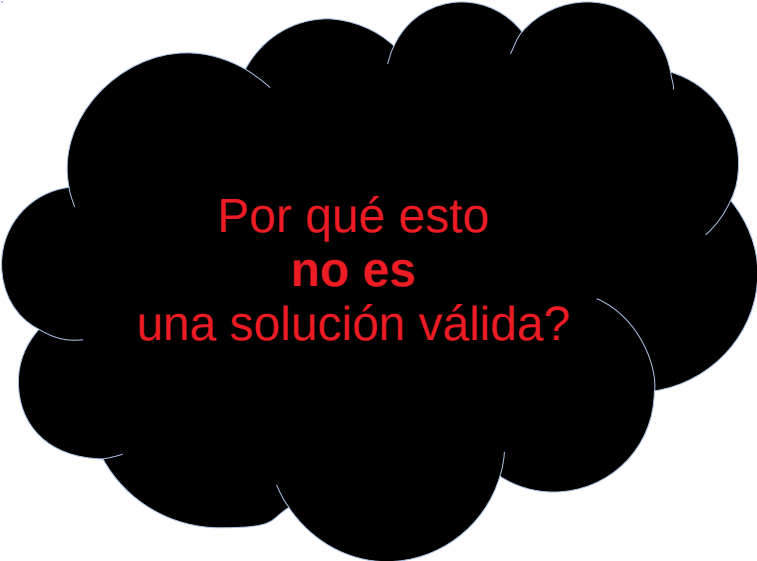
Solución aparente al problema anterior

```
public class Caja
{
    private int valor;
    private boolean disponible = false;

    public Caja()
    {
        super();
    }

    public synchronized void meter(int nv)
    {
        System.out.println("intentando meter el valor: " + nv);
        if (!disponible)
        {
            valor = nv;
            disponible = true;
            System.out.println("---> metido el valor: " + valor);
        }
    }

    public synchronized void sacar()
    {
        System.out.println("intentando sacar el valor disponible ");
        if (disponible)
        {
            System.out.println("---> sacado el valor: " + valor);
            valor = 0;
            disponible = false;
        }
    }
}
```



Por qué esto
no es
una solución válida?

Solución aparente al problema anterior

```
class Prod2 extends Thread
{
    Caja c;

    public Prod2(Caja nc)
    {
        c = nc;
    }

    public void run()
    {
        int i;
        for (i = 1; i <= 10; i++)
        {
            c.meter(i);
        }
    }
}
```

```
class Cons2 extends Thread
{
    Caja c;

    public Cons2(Caja nc)
    {
        c = nc;
    }

    public void run()
    {
        int i;
        for (i = 1; i <= 10; i++)
        {
            c.sacar();
        }
    }
}
```


Solución aparente al problema anterior

```
class ProdCons
{
    public static void main(String argum[])
    {
        Caja cj = new Caja();
        Prod2 p = new Prod2(cj);
        Cons2 c = new Cons2(cj);
        p.start();
        c.start();
    }
}
```

Por esto -->

Por esto -->

Después de sacar el valor 1, el consumidor ha seguido su ejecución en el bucle y como no se mete ningún valor por el consumidor, termina su ejecución, el productor introduce el siguiente valor (el 2), y sigue su ejecución hasta terminar el bucle; como el consumidor ya ha terminado su ejecución no se pueden introducir más valores y el programa termina.

[illegible]

Comunicación entre Threads: **wait**, **notifyAll**, **notify**

evitar interferencias

El mecanismo de bloqueo de synchronized es suficiente para evitar que los hilos interfieran entre sí, pero también necesitamos formas de comunicación entre hilos.

espera de un thread

En el caso del ejemplo, lo que hace falta es un mecanismo que produzca la espera del productor si la caja tiene algún valor (disponible) y otro que frene al consumidor si la caja está vacía (no disponible):

Comunicación entre Threads: **wait**, **notifyAll**, **notify**

Para este propósito el método **wait** deja un thread esperando hasta que alguna condición ocurra, y los métodos de notificación **notifyAll** y **notify** avisan a los threads que estan esperando que algo ha ocurrido que podría satisfacer la condición. Los métodos **wait** y los de notificación están definidos en **Object**.

```
synchronized void doWhenCondition()  
{  
    while (!condition)  
        wait();  
    //hace algo cuando la condición es verdadera  
}
```

Análisis del código: el hilo se pone a dormir

```
synchronized void doWhenCondition()  
{  
    while (!condition)  
        wait();  
    //hace algo cuando la condición es verdadera  
}
```

1.- Todo se ejecuta dentro de un código sincronizado

Si no fuera así, el estado del objeto no sería estable. Por ejemplo, si el método no fuera declarado `synchronized`, después de la sentencia `while`, no habría garantía que la condición se mantenga verdadera, otro thread podría cambiar la situación que la condición testea.

Análisis del código: el hilo se pone a dormir

```
synchronized void doWhenCondition()  
{  
    while (!condition)  
        wait();  
    //hace algo cuando la condición es verdadera  
}
```

2.- Suspende el thread y libera el bloqueo del recurso compartido

Uno de los aspectos más importantes de la definición de wait es que cuando suspende al thread, libera atómicamente el bloqueo sobre el objeto. Suceden juntas, de forma indivisible. Si no, habría riesgo de competencia: podría suceder una notificación después de que se liberara el bloqueo pero antes de que se suspendiera el hilo. La notificación no habría afectado al hilo, perdiéndose. Cuando el hilo se reinicia después de la notificación, el bloqueo se vuelve a adquirir atómicamente.

Análisis del código: el hilo se pone a dormir

```
synchronized void doWhenCondition()  
{  
    while (!condition)  
        wait();  
    //hace algo cuando la condición es verdadera  
}
```

3.- La condición de prueba debe estar siempre en un ciclo de repetición

La condición de prueba debería estar siempre en un bucle. Nunca debe asumirse que ser despertado significa que la condición se ha cumplido, ya que puede haber cambiado de nuevo desde que se cumplió. No debe cambiar un `while` por un `if`.

Análisis del código: se notifica a los hilos que despierten

```
synchronized void  
doWhenCondition()  
{  
    while (!condition)  
        wait();  
    //hace algo cuando condición es verdadera  
}
```

```
// se termina de usar  
// el recurso compartido  
notifyall();  
  
...  
}
```

4.- Conviene usar notifyAll() en vez de notify()

Al utilizar notifyAll se despiertan todos los hilos en espera (independientemente de su condición), y notify selecciona sólo un hilo para despertar (uno cualquiera). Si se despertara uno que no satisface la condición modificada, volvería a dormir, y el que hubiera debido despertarse nunca lo hizo.

El uso de notify es una optimización que sólo debe aplicarse cuando:

- Todos los hilos están esperando por la misma condición
- Sólo un hilo como mucho se puede beneficiar de que la condición se cumpla
- Esto es contractualmente cierto para todas las posibles subclases



Tread
consumidor



Único
Recurso
Compartido



Tread
productor



Tread
consumidor



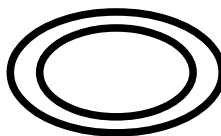
Tread
consumidor



Tread
consumidor



Tread
consumidor



Único
Recurso
Compartido



Tread
productor



Tread
consumidor



Tread
consumidor

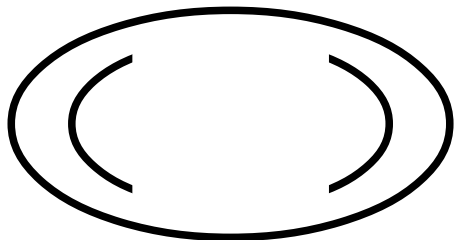


Tread
consumidor

```
synchronized void doWhenCondition()  
{  
    while (!condition)  
        wait();  
    //hace algo cuando la condición es verdadera  
}
```

Pensar

Ejemplo: Productor – Consumidor completo



El objetivo del ejemplo es generar un esquema de productor – consumidor, en donde cada hilo, compite por el uso del recurso compartido (CubbyHole).

En este caso, si el hilo no puede adquirir el recurso, en vez de terminar, espera hasta que esté disponible.

Para cada hilo, la condición para utilizar el recurso compartido es la opuesta.

Si el CubbyHole tiene un elemento, el hilo consumidor (Consumer) puede actuar y el productor (Producer) espera.

En el caso contrario, el hilo productor puede actuar y el consumidor espera.

Ejemplo: Productor – Consumidor completo

```
public class CubbyHole
{
    private int contents;
    private boolean available = false;

    public synchronized int get()
    {
        while (available == false)
        {
            try
            {
                wait();
            }
            catch (InterruptedException e)
            {
            }
        }
        available = false;
        notifyAll();
        return contents;
    }
}
```

```
public synchronized void put(int value)
{
    while (available == true)
    {
        try
        {
            wait();
        }
        catch (InterruptedException e)
        {
        }
    }
    contents = value;
    available = true;
    notifyAll();
}
```

Ejemplo: Productor – Consumidor completo



Las otras clases (los hilos) no sufren cambios

La sentencia **synchronized**

No es lo mismo que el método sincronizado. Puede usarse para ámbitos menores que la invocación completa del método.

```
metodo(...)  
{  
    ...  
    synchronized(ref a obj)  
    {  
        sentencias  
    }  
    ...  
}
```

Tiene dos partes:

- Un objeto cuyo bloqueo se va a adquirir
- Un grupo de sentencias que se ejecuta cuando se adquiere el bloqueo

La sentencia **synchronized**

```
synchronized(ref a obj)  
{  
    sentencias  
}
```

Cuando se obtiene el bloqueo, se ejecutan las sentencias del bloque, y al final del bloque el bloqueo se libera.

Si se produce una excepción en el interior del bloque, el bloqueo también se libera.

NOTA: el *método synchronized* es simplemente una abreviatura de un método cuyo cuerpo está envuelto en una sentencia *synchronized* con una referencia *this*

La sentencia **synchronized**: ejemplo

Método (propio del Thread, No del recurso compartido) para sustituir los elementos de un array con sus valores absolutos, que se basa en la sentencia synchronized para controlar el acceso al arreglo.

```
public static void abs(int[] valores)
{
    synchronized(valores)
    {
        for(int i=0; i < valores.length; i++)
        {
            if(valores[i] < 0)
                valores[i] = - valores[i];
        }
    }
}
```

La sentencia **synchronized**: ejemplo

Esto es un ejemplo de lo que se conoce como sincronización de parte del cliente. Todos los clientes que utilizan el objeto compartido (arreglo en este caso) están de acuerdo en sincronizarse sobre ese objeto antes de manejarlo.

```
public static void abs(int[] valores)
{
    synchronized(valores)
    {
        for(int i=0; i < valores.length; i++)
        {
            if(valores[i] < 0)
                valores[i] = - valores[i];
        }
    }
}
```


La sentencia **synchronized**: ventajas y desventajas

La sentencia synchronized tiene ventajas sobre la sincronización de métodos:



- Definir una región de código sincronizada más pequeña que un método.
- Permite sincronizar sobre objetos diferentes de this.
- Bloqueo selectivo. Permite sincronizar entre sí, componentes de una clase. Permite una sincronización “de a grupos”.

PERO



Si se la usa como parte del código del cliente (Thread) se está delegando la responsabilidad del buen uso al Thread. El objeto no se puede “defender” dentro de un ambiente concurrente.



PERO PERO

Se puede usar la sentencia synchronized dentro de un recurso compartido también.

La sentencia **synchronized**: bloqueo selectivo

```
class GruposSeparados
{
    private double valA = 0.0;
    private double valB = 1.1;
    protected Object bloqueoA = new
    protected Object bloqueoB = new

    public double getA()
    {
        synchronized ( bloqueoA )
        {
            return valA;
        }
    }

    public void setA(double val)
    {
        synchronized ( bloqueoA )
        {
            valA = val;
        }
    }
}
```

```
    public double getB()
    {
        synchronized ( bloqueoB )
        {
            return valB;
        }
    }

    public void setB(double val)
    {
        synchronized ( bloqueoB )
        {
            valB = val;
        }
    }

    public double inicializar()
    {
        synchronized ( bloqueoA )
        {
            synchronized ( bloqueoB )
            {
                valA = valB = 0.0;
            }
        }
    }
}
```

La sentencia **synchronized**: bloqueo selectivo

```
class GruposSeparados
{
    private double valA = 0.0;
    private double valB = 1.1;
    protected Object bloqueoA = new
    protected Object bloqueoB = new

    public double getA()
    {
        synchronized ( bloqueoA )
        {
            return valA;
        }
    }

    public void setA(double val)
    {
        synchronized ( bloqueoA )
        {
            valA = val;
        }
    }
}
```

```
    public double getB()
    {
        synchronized ( bloqueoB )
        {
            return valB;
        }
    }

    public void setB(double val)
    {
        synchronized ( bloqueoB )
        {
            valB = val;
        }
    }

    public double inicializar()
    {
        synchronized ( bloqueoA )
        {
            synchronized ( bloqueoB )
            {
                valA = valB = 0.0;
            }
        }
    }
}
```

Permite sincronizar en partes separadas.

O sea, la sincronización sobre A no afecta a los métodos que están sincronizados sobre B (y viceversa)

Semáforos



Un semáforo es un objeto que permite o impide el **paso** de otros objetos.

Un semáforo contiene un número de **permisos** que son los que los objetos intentan obtener para realizar alguna operación. Tiene un contador interno en el que lleva la cuenta de la cantidad de permisos que fueron otorgados.

Semáforos



Los métodos troncales de un semáforo son **acquire()** y **release()**. El primero intenta obtener un permiso del semáforo, si hay uno disponible, termina al instante; de lo contrario, es bloqueado hasta que haya un permiso. Por otra parte, release libera un permiso y lo devuelve al semáforo.

Semáforos parciales y imparciales Un semáforo parcial (por defecto) no da garantías sobre quién recibirá un permiso para liberarse. Si es imparcial otorga el permiso según el orden de bloqueo.

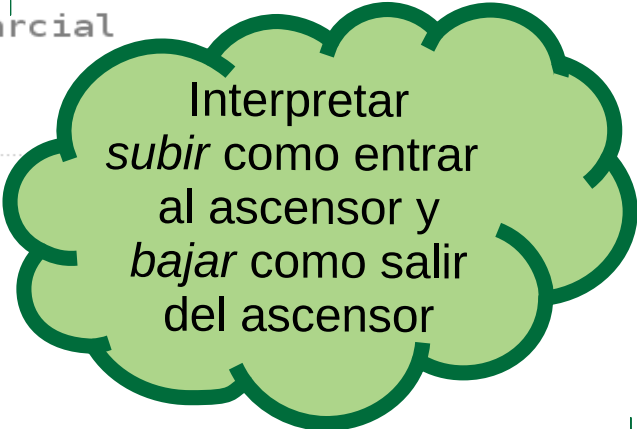
Semaforos: ejemplo simple

```
public class Ascensor
{
    private Semaphore semaforo;

    public Ascensor()
    {
        //un semáforo con capacidad para 10 permisos, imparcial
        semaforo = new Semaphore(10, true);
    }

    public void subir()
    {
        try
        {
            semaforo.acquire();
        }
        catch (InterruptedException ie)
        {
            System.out.println("La persona se arrepintió de subir al ascensor");
        }
    }

    public void bajar()
    {
        semaforo.release();
    }
}
```



Interpretar
subir como entrar
al ascensor y
bajar como salir
del ascensor

Semaforos: ejemplo simple

```
public class Ascensor
{
    private Semaphore semaforo;

    public Ascensor()
    {
        //un semáforo con capacidad para 10 permisos, imparcial
        semaforo = new Semaphore(10, true);
    }

    public void subir()
    {
        try
        {
            semaforo.acquire();
        }
        catch (InterruptedException ie)
        {
            System.out.println("La persona se arrepintió de subir al ascensor");
        }
    }

    public void bajar()
    {
        semaforo.release();
    }
}
```

El ascensor simplemente, delega a un semáforo la responsabilidad de llevar la cuenta de cuántas personas hay.

Notemos que el método `acquire` declara arrojar la excepción `InterruptedException`. Esto se debe a que, mientras el hilo está bloqueado esperando un permiso del semáforo, quizá sea interrumpido (usando el método `interrupt` de la clase `Thread`), y debemos capturar la excepción, en este caso, implicando que se rompió la espera.

Semaforos: ejemplo Productor - Consumidor



Desarrollar el ejemplo
del productor –
consumidor con
semáforos