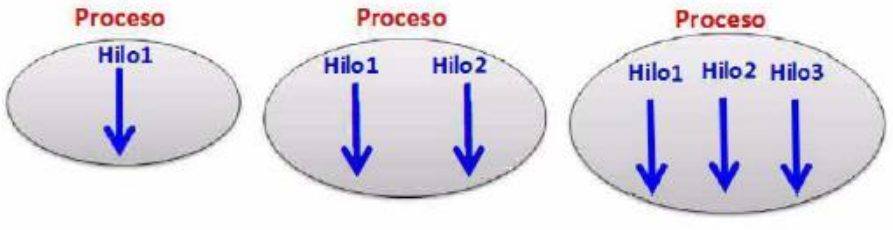
UNIDAD 2

PROGRAMACIÓN MULTIHILO

HILO

Un hilo (hebra, thread en inglés) es una secuencia de código en ejecución dentro del contexto de un proceso. Los hilos no pueden ejecutarse ellos solos, necesitan la supervisión de un proceso padre para ejecutarse. Dentro de cada proceso hay varios hilos ejecutándose.

Los hilos comparten el espacio de memoria, mientras que los procesos generalmente mantienen su propio espacio de memoria y entorno de operaciones. Por ello a los hilos se les conoce a menudo como procesos ligeros.



Podemos usar los hilos para diferentes aplicaciones: para realizar programas que tengan que realizar varias tareas simultáneamente. Por ejemplo, un programa que controla sensores en una fábrica, cada sensor puede ser un hilo independiente y recoge un tipo de información; y todos deben controlarse de forma simultánea.

CLASES PARA LA CREACIÓN DE HILOS

En Java existen dos formas para crear hilos: extendiendo la clase Thread o implementando la interfaz Runnable. Ambas son parte del paquete java.lang.

CLASE THREAD

La forma más simple de añadir funcionalidad de hilo a una clase es extender la clase Thread. Se debe sobrescribir el método run() con las acciones que el hilo debe desarrollar. La forma general de declarar un hilo extendiendo Thread es la siguiente:

```
class NombreHilo extends Thread {
        //propiedades, constructores y métodos de la clase
        public void run() {
                //acciones que lleva a cabo el hilo
        }
}
```

Para crear un objeto hilo con el comportamiento de NombreHilo:

```
NombreHilo h = new NombreHilo();
```

Y para iniciar su ejecución utilizamos el método start():

```
h.start();
```

```java
public class PrimerHilo extends Thread {
    private int x;

    PrimerHilo(int x) {
        this.x = x;
    }

    public void run() {
        for (int i = 0; i < x; i++)
            System.out.println("En el Hilo... " + i);
    }

    public static void main(String[] args) {
        PrimerHilo p = new PrimerHilo(10);
        p.start();
    }// main

}
```

Cuando se crea un nuevo hilo o thread mediante el método new(), no implica que el hilo ya se pueda ejecutar. Para que el hilo se pueda ejecutar, debe estar en el estado "Ejecutable", y para conseguir ese
estado es necesario iniciar o arrancar el hilo mediante el método start() de la clase thread()

El método start() realiza las siguientes tareas:

- Crea los recursos del sistema necesarios para ejecutar el hilo.
- Se encarga de llamar a su método run() y lo ejecuta como un subproceso nuevo e independiente.

Es por esto último que cuando se invoca a start() se suele decir que el hilo está "corriendo", pero esto no significa que el hilo esté ejecutándose en todo momento. Un hilo "Ejecutable" puede estar "Preparado" o "Ejecutándose" según tenga o no asignado tiempo de procesamiento.

Algunas consideraciones importantes que debes tener en cuenta son las siguientes:

- Puedes invocar directamente al método run(), por ejemplo poner hilo1.run(); y se ejecutará el código asociado a run() dentro del hilo actual (como cualquier otro método), pero no comenzará un nuevo hilo como subproceso independiente.
- 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, no se conoce la secuencia en la que serán ejecutadas la instrucciones del programa.

THREAD

MÉTODOS	MISIÓN
start()	Hace que el hilo comience la ejecución; la máquina virtual de Java llama al método **run()** de este hilo.
boolean isAlive()	Comprueba si el hilo está vivo
sleep(long mils)	Hace que el hilo actualmente en ejecución pase a dormir temporalmente durante el número de milisegundos especificado. Puede lanzar la excepción *InterruptedException*.
run()	Constituye el cuerpo del hilo. Es llamado por el método **start()** después de que el hilo apropiado del sistema se haya inicializado. Si el método **run()** devuelve el control, el hilo se detiene. Es el único método de la interfaz **Runnable**.
String toString()	Devuelve una representación en formato cadena de este hilo, incluyendo el nombre del hilo, la prioridad, y el grupo de hilos. Ejemplo: Thread[HILO1,2,main]
long getId()	Devuelve el identificador del hilo.
void yield()	Hace que el hilo actual de ejecución pare temporalmente y permita que otros hilos se ejecuten.
String getName()	Devuelve el nombre del hilo.
setName(String name)	Cambia el nombre de este hilo, asignándole el especificado como argumento.
int getPriority()	Devuelve la prioridad del hilo.
setPriority(int p)	Cambia la prioridad del hilo al valor entero p.
void interrupt()	Interrumpe la ejecución del hilo
boolean interrupted()	Comprueba si el hilo actual ha sido interrumpido.
Thread currentThread()	Devuelve una referencia al objeto hilo que se está ejecutando actualmente.
boolean isDaemon()	Comprueba si el hilo es un hilo Daemon. Los hilos daemon o demonio son hilos con prioridad baja que normalmente se ejecutan en segundo plano. Un ejemplo de hilo demonio que está ejecutándose continuamente es el recolector de basura (*garbage collector*).
setDaemon(boolean on)	Establece este hilo como hilo Daemon, asignando el valor *true*, o como hilo de usuario, pasando el valor *false*.
void stop()	Detiene el hilo. Este método está en desuso.
Thread currentThread()	Devuelve una referencia al objeto hilo actualmente en ejecución.
int activeCount()	Este método devuelve el número de hilos activos en el grupo de hilos del hilo actual.
Thread.State getState()	Devuelve el estado del hilo: NEW, RUNNABLE, BLOCKED, WAITING, TIMED_WAITING, TERMINATED

Ejercicio HiloEjemplo1:

Crear una clase llamada HiloEjemplo1 que extienda la clase Thread.

- Definir el constructor, el método run() y el método main()
- En el método run() se visualizará un mensaje donde se muestre el nombre del hilo y un contador.
- En el constructor se usará una variable para determinar el nombre del hilo. Esa variable se la debemos pasar la clase base Thread.
- En el método main() se crearán y lanzarán 3 hilos.

En el siguiente ejemplo usamos algunos métodos de la clase Thread

```
public class HiloEjemplo21 extends Thread {

  public void run() {
     System.out.println("Dentro del Hilo  : " + Thread.currentThread().getName());
  }
  //

  public static void main(String[] args) {

    HiloEjemplo21 h = new HiloEjemplo21(); //crear hilo
    h.setName("HILO 1");    //damos nombre al hilo
    h.start();
    System.out.println("Informacion del " + h.getName() + ": "+ h.toString());

    System.out.println("1 HILO CREADO...");
    System.out.println("Hilos activos: " +Thread.activeCount());
  }//

}
```

Todo hilo de ejecución en Java debe formar parte de un grupo. Por defecto, si no se especifica ningún grupo en el constructor, los hilos serán miembros del grupo main, que es creado por el sistema cuando arranca la aplicación Java.

La clase ThreadGroup se utiliza para manejar grupos de hilos en las aplicaciones Java. La clase Thread proporciona constructores en los que se puede especificar el grupo del hilo que se está creando en el mismo momento de instanciarlo.

```java
public class HiloEjemplo2Grupos extends Thread {

    private ThreadGroup grupo;
    private String nombre;

    public void run() {
        System.out.println("Informacion del hilo: " + Thread.currentThread().toString());

        for (int i = 0; i < 1000; i++)
            i++;

        System.out.println(Thread.currentThread().getName() + " Finalizando la ejecución.");
    }// run

    public static void main(String[] args) {

        Thread.currentThread().setName("Principal");
        System.out.println(Thread.currentThread().getName());
        System.out.println(Thread.currentThread().toString());

        ThreadGroup grupo = new ThreadGroup("Grupo de hilos");
        HiloEjemplo2Grupos h = new HiloEjemplo2Grupos();

        Thread h1 = new Thread(grupo, h, "Hilo 1");
        Thread h2 = new Thread(grupo, h, "Hilo 2");
        Thread h3 = new Thread(grupo, h, "Hilo 3");

        h1.start();
        h2.start();
        h3.start();
    }//
}
```

Ejercicio Actividad2_1

Crea dos clases (hilos) Java que extiendan la clase Thread. Uno de los hilos debe visualizar en pantalla en un bucle infinito la palabra TIC y el otro hilo la palabra TAC. Debe darnos tiempo a ver las palabras que se visualizan cuando lo ejecutemos. Crea después la función main() que haga uso de los hilos anteriores.

Ejercicio Cronómetro:

Crea un programa en Java que tenga la funcionalidad de un cronómetro usando hilos.

Ejercicio Ejercicio1:

Crear una clase que extienda Thread cuya única funcionalidad sea visualizar el mensaje “Hola mundo” y el identificador del hilo. Crea un programa Java que visualice el mensaje anterior 5 veces creando para ello 5 hilos diferentes usando la clase creada anteriormente.

CLASE RUNNABLE

Para añadir la funcionalidad de hilo a una clase que deriva de otra clase (por ejemplo, un Applet), siendo ésta distinta de Thread, se utiliza la interfaz Runnable. Esta interfaz añade la funcionalidad de hilo a una clase con solo implementarla. La interfaz Runnable proporciona un único método, el método run().

```
class NombreHilo implements Runnable {
        //propiedades, constructores y métodos de la clase
        public void run() {
                //acciones que lleva a cabo el hilo
        }
}
```

Para crear un objeto hilo con el comportamiento de NombreHilo escribo lo siguiente:

NombreHilo h = new NombreHilo();

Y para iniciar su ejecución utilizamos el método start():

Thread h1 = new Thread(h).
h1.start()

El siguiente ejemplo declara la clase PrimerHiloR que implementa la interfaz Runnable, en el método run() se indica la funcionalidad del hilo, en este caso es pintar un mensaje y visualizar el identificador del hilo actualmente en ejecución.

```
public class PrimerHiloR implements Runnable {
    public void run() {
        System.out.println("Hola desde el Hilo! " +
           Thread.currentThread().getId());
    }

    public static void main(String[] args) {
        //Primer hilo
        PrimerHiloR hilo1 = new PrimerHiloR();
        new Thread(hilo1).start();

        //Segundo hilo
        PrimerHiloR hilo2 = new PrimerHiloR();
        Thread hilo = new Thread(hilo2);
        hilo.start();

        //Tercer Hilo
        new Thread(new PrimerHiloR()).start();
    }
}//
```

Ejercicio Ejercicio2

Crea una clase que implemente la interfaz Runnable cuya única funcionalidad sea visualizar el mensaje "Hola mundo" seguido de una cadena que se recibirá en el constructor y seguido del identificador del hilo.
Crea un programa Java que visualice el mensaje anterior 5 veces creando para ello 5 hilos diferentes usando la clase creada anteriormente. Hacer que antes de visualizar el mensaje el hilo espere un tiempo proporcional a su identificador;

Vamos a hacer un applet que implemente Runnable y ver cómo usar un hilo en un applet para realizar una tarea repetitiva, por ejemplo mostrar la hora con los minutos y segundos: HH:MM:SS

La estructura general de applet que implemente Runnable es algo similar a los siguiente:

```
public class AppletThread extends Applet implements Runnable {
        private Thread hilo null;

        public void init() {
        }

        public void start() {
                if (hilo == null) {
                hilo = new Thread(this); // crea el hilo
                hilo.start(); // lanza el hilo
                }
        }

        public void run() {
                Thread hiloActual = Thread.currentThread();
                while (hilo == hiloActual)  {
                        // tarea repetitiva
        }

        public void stop(){
                hilo = null;
        }
        public void paint(Graphics g) {

        }
}
```

• init(): con instrucciones para inicializar el applet, este método es llamado una vez cuando se carga el applet.
• start(): parecido a init() pero con la diferencia de que es llamado cuando se reinicia el applet.
• paint(): que se encarga de mostrar el contenido del applet; se ejecuta cada vez que se tenga que redibujar.
• stop(): es invocado al ocultar el applet, se utiliza para detener hilos.

Ejercicio Reloj.java

A partir de la estructura RelojIncompleto.java, donde tenemos la estructura y diseño de nuestro applet, completar el código en los métodos start() y run() para que nuestro applet nos muestre la hora en formato HH:MM:SS usando un hilo repetitivo. Usar clases Calendar y SimpleDateFormat.

ESTADOS DE UN HILO

NEW (NUEVO)

Es el estado cuando se crea un objeto hilo con el operador *new*. El hilo todavía no se está ejecutando.

RUNNABLE (EJECUTABLE)

Un hilo pasa a este estado cuando se invoca al método start(). El sistema debe asignar tiempo CPU al hilo. Por tanto, en este estado un hilo puede estar o no en ejecución.

DEAD (MUERTO)

Un hilo puede morir por varias razones: de muerte natural, porque el método run() finaliza con normalidad; y repentinamente debido a alguna excepción no capturada en el método run().

El método stop() sirve para matar un hilo, aunque actualmente está en desuso porque no libera de forma inmediata los recursos del sistema. Para detener un hilo de forma segura, se puede usar una variable booleana, por ejemplo, de la siguiente manera:

```
public class HiloEjemploDead extends Thread{

    private boolean stopHilo = false;

    public void pararHilo() {
        stopHilo = true;
    }

    public void run() {
        while (!stopHilo) {
            System.out.println("En el hilo");
        }
    }

    public static void main (String[] args) {
        HiloEjemploDead h = new HiloEjemploDead();
        h.start();
        for (int i = 0; i < 100000; i++);
        h.pararHilo();
    }

}
```

BLOCKED (BLOQUEADO)

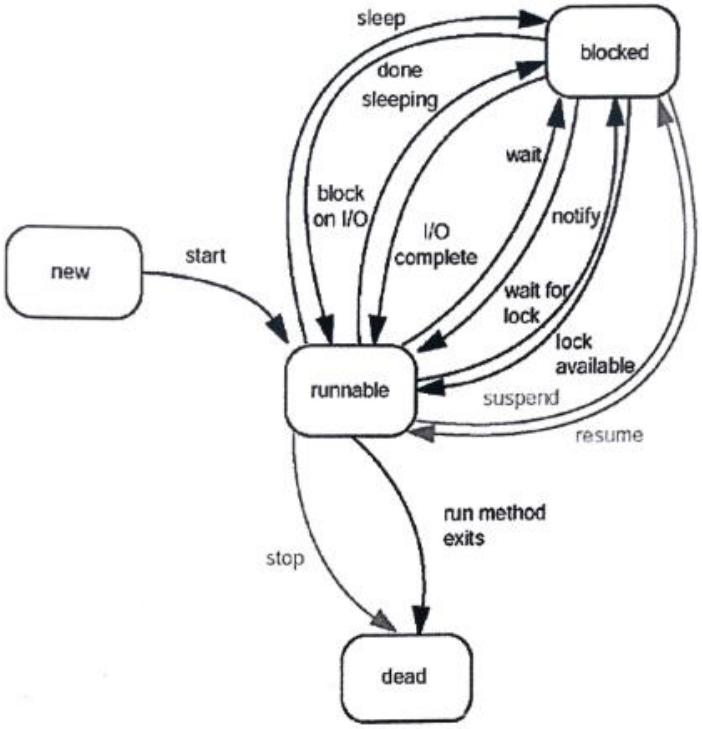
En este estado, el hilo podría ejecutarse, pero hay algo que lo evita. Un hilo pasa a este estado cuando ocurre alguna de las siguientes acciones:

- Se llama al método sleep() del hilo.

- El hilo está esperando a que se complete una operación de E/S.

- El hilo llama al método wait(). El hilo no volverá a estado ejecutable hasta que reciba los mensajes notify() o notifyAll().

- Se llama al método suspend() del hilo. El hilo no volverá a estado ejecutable hasta que reciba el mensaje resume(). Obsoletos



Con el método getState() podemos comprobar el estado de un hilo. Este método nos devuelve un valor constante que puede ser:

-NEW: El hilo aún no se ha iniciado
-RUNNABLE: El hilo se está ejecutando
-BLOCKED: El hilo está bloqueado
-WAITING: El hilo está esperando indefinidamente hasta que otro realice una acción.
-TIMED_WAITING: El hilo está esperando hasta que otro realice una acción un tiempo de espera especificado.
-TERMINATED: El hilo ha finalizado.

GESTIÓN DE HILOS

CREACCIÓN DE HILOS

Thread // Runnable

SUSPENSIÓN DE HILOS

El método sleep() sirve para detener un hilo un número de milisegundos. Realmente el hilo no se detiene, sino que se queda "dormido" el número de milisegundos que indiquemos.

El método suspend() permite detener la actividad del hilo durante un intervalo de tiempo indeterminado. Para volver a activar el hilo se necesita invocar al método resume(). Ambos métodos están en desuso porque puede producir situaciones de interbloqueos. Por ejemplo, si un hilo está bloqueando un recurso y este hilo se suspende puede dar lugar a que otros hilos que esperaban el recurso queden "congelados" ya que el hilo suspendido mantiene los recursos bloqueados.

Para suspender de **forma segura** el hilo se debe introducir en el hilo una variable, y comprobar su valor dentro del método run().

```
class MyHilo extends Thread {
        private SolicitaSuspender suspender = new SolicitaSuspender();
        public void Suspende() {    //petición de SUSPENDER
            suspender.set(true);
        }
        public void Reanuda() {    //petición de CONTINUAR
            suspender.set(false);
        }
        public void run() {
            try {
              while (haya trabajo por hacer) {
                 …
                 suspender.esperandoParaReanudar(); //comprobar
                …
               }
            } catch (InterruptedException exception) {}
        }}

public class SolicitaSuspender {
    private boolean suspender;
        public synchronized void set(boolean b) {
        suspender = b;
        notifyAll () ;
        }
        public synchronized void esperandoParaReanudar() throws
                                                        InterruptedException {
        while (suspender)
           wait();
        }
```

La clase SolicitaSuspender tiene el método set() que da el valor true o false a la variable y llama al método notifyAII(), el cuál notifica a todos los hilos que esperan (han ejecutado un wait()) un cambio de estado sobre el objeto. En el método esperandoParaReanudar() se hace un wait() cuando el valor de la variable es true, el método wait() hace que el hilo espere hasta que le llegue un notify() o un notifyAII();

El método wait() sólo puede ser llamado desde dentro de un método sincronizado (synchronized). Estos tres métodos: wait(), notify() y notifyAII(), se usan en sincronización de hilos. Forman parte de la clase **Object.**

Método	Tipo de retorno	Descripción
wait()	*void*	Implementa la operación *wait*. El hilo espera hasta que otro hilo invoque *notify* o *notifyAll()*
notify()	*void*	Implementa la operación *signal*. Desbloquea un hilo que esté esperando en el método *wait()*
notifyAll()	*void*	Despierta a todos los hilos que estén esperando para que continúen con su ejecución. Todos los hilos esperando por *wait* reanudan su ejecución.

PARADA DE HILOS

El método stop() detiene la ejecución de un hilo de forma permanente, impidiendo que está se pueda volver a reanudar o ejecutar con el método start(). Éste método, al igual que suspend(), resume() y destroy(), son obsoletos con el fin de evitar interbloqueos entre hilos. Se recomienda el uso de una variable como se vio anteriormente.

El método isAlive() devuelve true si el hilo está vivo.

El método interrupt() envía una petición de interrupción a un hilo. Si el hilo se encuentra bloqueado por una llamada a sleep() o wait() se lanza una excepción InterruptedException. El método isInterrupted() devuelve true si el hilo ha sido interrumpido.

```java
public class HiloEjemploInterrup extends Thread {
  public void run() {
    try {
      while (!isInterrupted()) {
        System.out.println("En el Hilo");
        Thread.sleep(10);
      }
      } catch (InterruptedException e) {
         System.out.println("HA OCURRIDO UNA EXCEPCIÓN");
      }

      System.out.println("FIN HILO");
   }//run


   public void interrumpir() {
       interrupt();
   }//interrumpir

   public static void main(String[] args) {
    HiloEjemploInterrup h = new HiloEjemploInterrup();
    h.start();
    for(int i=0; i<1000000000; i++) ;//no hago nada
    h.interrumpir();
  }//
}//
```

GESTIÓN DE PRIORIDADES

Los hilos heredan la prioridad del padre en Java, pero este valor puede ser cambiado con el método setPriority() y con getPriority() podemos saber la prioridad de un hilo.

El valor de la prioridad varía entre 1 y 10; MIN_PRIORITY (1) MAX_PRIORITY(10) NORM_PRIORITY(5) … el planificador elige el hilo en función de su prioridad. Si dos hilos tienen la misma prioridad realiza un round-robin, es decir de forma cíclica va alternando los hilos.

Ejercicio EjemploHiloPrioridad1.java -> clase de mañana en este ejemplo
Dada la clase HiloPrioridad1.java, crear un programa que cree 3 hilos de esa clase dando a cada uno MIN_PRIORITY, MAX_PRIORITY y NORM_PRIORITY. Ejecutar durante un tiempo razonable (10 segundos)

SINCRONIZACIÓN DE HILOS

Los threads se comunican principalmente mediante el intercambio de información a través de variables y objetos en memoria. Todos los threads pertenecen al mismo proceso y por tanto pueden acceder a toda la memoria asignada a dicho proceso y utilizar las variables y objetos del mismo para compartir información, siendo este el método de comunicación más eficiente.

Sin embargo, cuando varios hilos manipulan concurrentemente objetos conjuntos, puede llevar a resultados erróneos o a la paralización de la ejecución. La solución es la sincronización.

PROBLEMAS DE SINCRONIZACIÓN

CONDICIÓN DE CARRERA

Se dice que existe una condición de carrera si el resultado de la ejecución de un programa depende del orden concreto en que se realicen los accesos a memoria.

Ejemplo: un hilo sumador (contador++) y otro restador (contador--) se ejecutan al mismo tiempo sobre la misma variable. Si contador=10, podemos obtener en un momento dado contador=11 o contador=9.

INCONSISTENCIA DE MEMORIA

Se produce cuando diferentes hilos tienen una visión diferente de lo que debería ser un mismo dato. Las causas son complejas como por ejemplo la no liberación de datos obsoletos o el desbordamiento de buffer.

Ejemplo: dos hilos, sumador (contador++) e impresor (pinta contador); su ejecución en un hilo no genera problemas, pero en 2 hilos distintos sin sincronización programada el resultado puede variar.

INANICIÓN

Se conoce como inanición al fenómeno que tiene lugar cuando a un proceso o hilo se le deniega continuamente el acceso a un recurso compartido. Este problema se produce cuando un proceso nunca llega a tomar el control de un recurso debido a que el resto de procesos o hilos siempre toman el control antes que él por diferentes motivos. La inanición puede ocurrir tanto por prioridad como por la propia codificación.

INTERBLOQUEO

Se produce cuando dos o más procesos o hilos están esperando indefinidamente por un evento que solo puede generar un proceso o hilo bloqueado. En un ejemplo del mundo real, un bloqueo activo ocurre, por ejemplo, cuando dos personas se encuentran en un pasillo avanzando en sentidos opuestos y cada una trata de ser amable moviéndose a un lado para dejar a la otra persona pasar. Si se mueven ambas de lado a lado, encontrándose siempre en el mismo lado, se están moviendo, pero ninguna podrá avanzar.

SOLUCIONES A LA SINCRONIZACIÓN

CONDICIONES DE BERNSTEIN

Visto en la Unidad 1

OPERACIONES ATÓMICAS

Cuando una operación siempre se ejecutará de forma íntegra en un paso. Es decir, continuada sin ser interrumpida, por lo que ningún otro proceso o hilo puede leer o modificar datos relacionados mientras se esté realizando la operación. Es como si se realizara en un único paso. Es complicado implementarlo. Una forma de codificarlo sería declarar las variables como *volatile* (las variables se actualizan en memoria directamente)

Ejemplo: operación de sacar dinero en un cajero automático.

SECCIÓN CRÍTICA

Se denomina sección crítica a una región de código en la cual se accede de forma ordenada a variables y recursos compartidos. Cuando un proceso o hilo está ejecutando su sección crítica, ningún otro puede ejecutar su correspondiente sección crítica. Esto significa:

- Si otro proceso quiere ejecutar su sección crítica, se bloqueará hasta que el primer proceso finalice la ejecución de su sección crítica.
- Se establece una relación antes-después en el orden de ejecución de la sección crítica. Por ello, los cambios de los datos son visibles a todos los procesos o hilos.

El problema de la sección crítica consiste en diseñar un protocolo que permita a los procesos cooperar. Para ello, cualquier implementación de una sección crítica debe cumplir:

- Exclusión mutua: Su principal característica. Si un proceso está ejecutando su sección crítica, ningún otro tiene acceso a ella
- Progreso: cuando ningún proceso está ejecutando su sección crítica y en un momento dado varios necesitan acceder, la decisión debe ser ágil y se basara en la prioridad de cada uno.
- Espera limitada: debe existir un limitado de veces que se permite a otros procesos acceder a su sección crítica después de que otro proceso haya solicitado entrar en la suya.

MECANISMOS PARA LA SINCRONIZACIÓN

SEMÁFOROS

Es uno de los sincronizadores más clásicos para establecer zonas de sección crítica. Los semáforos más sencillos son binarios. Para entrar en una zona crítica un thread debe adquirir el derecho de acceso, y al salir lo libera.

```
Semaphore semaphore = new Semaphore(1);

semaphore.acquire();
    // zona crítica
semaphore.release();
```

Por solidez, nunca debemos olvidar liberar un semáforo al salir de la zona crítica. El código previo lo hace si salimos normalmente; pero si se sale de forma abrupta (return o excepción) entonces el semáforo no se liberaría. Es por ello, que normalmente se sigue este patrón:

```
semaphore.acquire();
try {
    // zona crítica
} finally {
    semaphore.release();
}
```

Más general, el semáforo puede llevar cuenta de N permisos. Los threads solicitan algunos permisos; si los hay, los retiran y siguen; si no los hay, quedan esperando a que los haya. Cuando ha terminado, el thread devuelve los permisos.

Método	Tipo de retorno	Descripción
Semaphore(int valor)	*void*	Inicialización del semáforo. Indica el valor inicial del semáforo antes de comenzar su ejecución
acquire()	*void*	Implementa la operación *wait*
release()	*void*	Implementa la operación *signal*. Desbloquea un hilo que esté esperando en el método *wait*()

BLOQUES Y MÉTODOS SINCRONIZADOS

Las operaciones se hacen de forma íntegra, es decir, si estamos realizando la suma nos asegurarnos que nadie realice la resta hasta que no terminemos la suma. Esto se puede lograr añadiendo la palabra synchronized a la parte de código que queramos que se ejecute de forma atómica. Java utiliza los bloques synchronized para implementar las regiones críticas. Podemos sincronizar bloques de la siguiente manera

```
class TwoMutex extends Thread {

            private Object o1 = new Object() ;
            private Object o2 = new Object() ;

            public void inc1() {
                synchronized(o1) {
                    //instrucciones
                }
            }

            public void inc2() {
                synchronized (o2) {
                    //instrucciones
                }
            }

            public void run(){
                inc1();
                inc2();
            }
}
```

Se debe evitar la sincronización de bloques de código y sustituirlas siempre que sea posible por la sincronización de métodos