¿Qué son los threads?

Todos los programadores conocen lo que es un proceso, la mayoría diría que es un programa en ejecución: tiene un principio, una secuencia de instrucciones y tiene un final.

Un thread es un flujo simple de ejecución dentro de un programa. Hasta el momento, todos los programas creados contenían un único thread, pero un programa (o proceso) puede iniciar la ejecución de varios de ellos concurrentemente. Puede parecer que es lo mismo que la ejecución de varios procesos concurrentemente a modo del fork() en UNIX, pero existe una diferencia. Mientras en los procesos concurrentes cada uno de ellos dispone de su propia memoria y contexto, en los threads lanzados desde un mismo programa, la memoria se comparte, utilizando el mismo contexto y recursos asignados al programa (también disponen de variables y atributos locales al thread).

Un thread no puede existir independientemente de un programa, sino que se ejecuta dentro de un programa o proceso.

Ejemplo

```
class UnThreadDosInstancias
{
    public static void main(String args[])
    {
        SiNoThread s = new SiNoThread("SI");
        SiNoThread n = new SiNoThread("NO");
        s.start();
        n.start();
    }
}

class SiNoThread extends Thread
{
    private String SiNo;
    static int Contador = 0;
    public SiNoThread(String s)
    {
        super();
        SiNo = s;
    }

    public void run()
    {
        int i;
        for (i = 1; i <= 20; i++)
        {
            System.out.print(++Contador + ":" + SiNo + " ");
        }
    }
}
```

Produce la siguiente salida:

```
1:SI 2:SI 3:SI 5:NO 6:NO 4:SI 8:SI
12:SI 13:SI 14:SI 15:SI 7:NO 17:NO
21:SI 22:SI 23:SI 24:SI 25:SI 18:NO
29:NO 30:NO 31:NO 32:NO 33:NO 34:NO
38:NO 39:NO 40:NO 9:SI 16:SI 26:NO
```

```
35:NO 10:SI 19:SI 27:NO 36:NO 11:SI
20:SI 28:NO 37:NO
```

En este caso se declaran dos instancias de una misma clase (SiNoThread) y se ejecutan concurrentemente. Cada una de ellas con sus propios atributos de objeto (String SiNo), pero comparten los atributos de clase (int Contador).

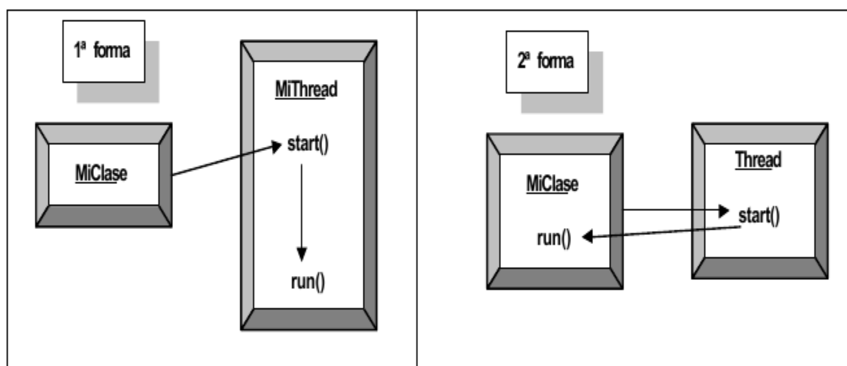
Puede comprobarse que el contador no ha funcionado todo lo correctamente que pudiera esperarse: 1,2,3,5,6,4,8, … Esto es debido a que se ha accedido concurrentemente a una misma zona de memoria sin que se produzca exclusión mutua.

Creación de threads.

Pueden crearse threads de dos formas distintas: declarando una subclase de la clase Thread o declarando una clase que implemente la interface Runnable y redefiniendo el método run() y start() de la interface.

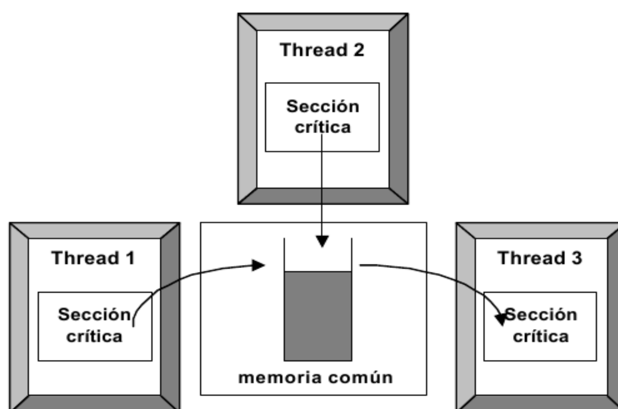
Se utilizará la primera forma, más evidente y sencilla, cuando la clase declarada no tenga que ser subclase de ninguna otra superclase.

Se utilizará la segunda forma cuando la clase declarada tenga que ser subclase de una superclase que no es subclase de Thread o implemente el interface Runnable.



Sección crítica

Se llama sección crítica a los segmentos de código dentro de un programa que acceden a zonas de memoria comunes desde distintos threads que se ejecutan concurrentemente



En Java, las secciones críticas se marcan con la palabra reservada synchronized. Aunque está permitido

marcar bloques de código más pequeños que un método como synchronized, para seguir una buena metodología de programación, es preferible hacerlo a nivel de método.

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

    public void run()
    {
        int i;
        long aux;
        for (i = 1; i <= 100000; i++)
        {
            contador.incrementa();
        }
        System.out.println("Contado hasta ahora: "
                + contador.getValor());
    }
}

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();
    }
}
```

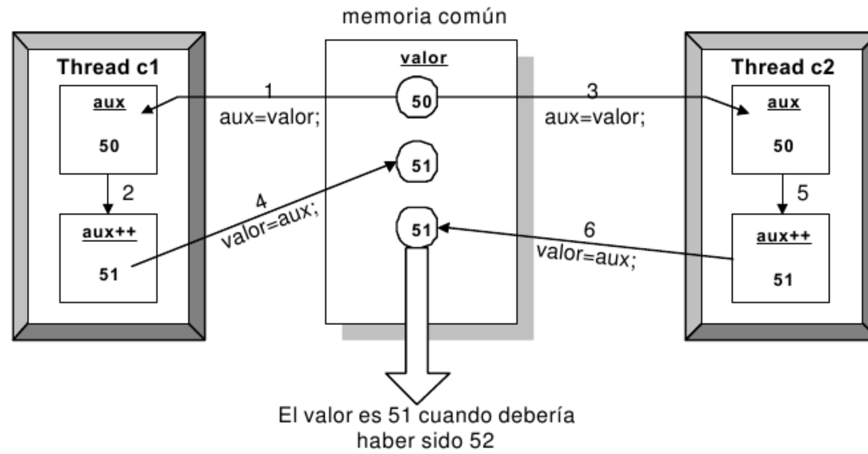
La salida por pantalla es:

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.

Lo que ha ocurrido en realidad para que se produzca este resultado equivocado es lo siguiente:



Monitores.

En Java, cada objeto que posee un método synchronized posee un único monitor para ese objeto.

Cuando un thread está ejecutando un método synchronized de un objeto, se convierte en el propietario del monitor, evitando que cualquier otro thread ejecute ningún otro método synchronized sobre ese mismo objeto. Esto no impide que el propietario del monitor ejecute otro método synchronized de ese mismo objeto, adquiriendo de nuevo el monitor (llamada a un método synchronized desde otro método synchronized).

```
class Caja
{
    private int valor;
    public synchronized void meter(int nv)
    {
        valor = nv;
        System.out.println("metido el valor: " + valor);
    }

    public synchronized void sacar()
    {
        System.out.println("sacado el valor: " + valor);
        valor = 0;
    }
}

class Productor extends Thread
{
    Caja c;
    public Productor(Caja nc)
    {
        c = nc;
    }

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

```
            c.meter(i);
        }
    }
}

class Consumidor extends Thread
{
    Caja c;

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

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

class ProdCons
{

    public static void main(String argum[])
    {
        Caja cj = new Caja();
        Productor p = new Productor(cj);
        Consumidor c = new Consumidor(cj);
        p.start();
        c.start();
    }
}
```

Tanto el thread 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 meter y sacar como synchronized. Pero esto no es suficiente para que el programa funcione correctamente, ya que el productor puede almacenar varios valores antes de que el consumidor extraiga el valor o, también, que el consumidor intente sacar varios valores consecutivamente, por lo que la salida por pantalla podría ser la siguiente:

```
metido el valor: 1
sacado el valor: 1
sacado el valor: 0
sacado el valor: 0
metido el valor: 2
metido el valor: 3
sacado el valor: 3
sacado el valor: 0
sacado el valor: 0
metido el valor: 4
metido el valor: 5
sacado el valor: 5
sacado el valor: 0
metido el valor: 6
metido el valor: 7
metido el valor: 8
sacado el valor: 8
sacado el valor: 0
metido el valor: 9
metido el valor: 10
```

De alguna forma habría que asegurar que no se meta ningún valor si la caja está llena y que no se saque ningún valor si la caja está vacía.

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

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

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

class Productor extends Thread
{
    Caja c;

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

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

class Consumidor extends Thread
{
    Caja c;

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

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

```
        {
            c.sacar();
        }
    }
}

class ProdCons2
{

    public static void main(String argum[])
    {
        Caja cj = new Caja();
        Productor p = new Productor(cj);
        Consumidor c = new Consumidor(cj);
        p.start();
        c.start();
    }
}
```

La salida por pantalla es:

```
metido el valor: 1
sacado el valor: 1
metido el valor: 2
```

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.

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, and notify)

La sincronización alcanza para evitar que varios threads interfieran entre si, pero se necesita un medio para comunicarlos entre si. 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 puede satisfacer la condición. Los métodos wait y los de notificación están definidos en Object.

Existe un patrón estandar que es importante usar con wait y notify. Se debe escribir así:

```
synchronized void doWhenCondition()
{
     while (!condition)
        wait();

     //hace algo cuando la condición es verdadera

}
```

Suceden un número de cosas:

•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, despues de la sentencia while, no habría garantía que la condición se mantenga verdadera, otro threas podría cambiar la situación que la condición testea.

•Uno de los aspecto 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 despues de que se liberara el bloqueo pero antes de que se suspendiera el hilo. La notificación no habría afectado el hilo, perdiéndose. Cuando el hilo se reinicia despues de la notificación, el bloqueo se vuelve a adquirir atómicamente.

•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.

Por otra parte, los métodos de notificación son invocados por código sincronizado que cambia una o más condiciones por las que algún otro hilo puede estar esperando.

Al utilizar notifiAll 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

```
public class Producer extends Thread
{
   private CubbyHole cubbyhole;
   private int number;

   public Producer(CubbyHole c, int number)
   {
      cubbyhole = c;
      this.number = number;
   }

   public void run()
   {
      for (int i = 0; i < 10; i++)
      {
         cubbyhole.put(i);
         System.out.println("Producer #" + this.number + " put : " + i);
         try
         {
            sleep((int) (Math.random() * 100));
         }
         catch (InterruptedException e)
```

```
            { }
        }
    }
}

public class Consumer extends Thread
{
    private CubbyHole cubbyhole;
    private int number;

    public Consumer(CubbyHole c, int number)
    {
        cubbyhole = c;
        this.number = number;
    }

    public void run()
    {
        int value = 0;
        for (int i = 0; i < 10; i++)
        {
            value = cubbyhole.get();
            System.out.println("Consumer #" + this.number + " got : " + value);
        }
    }
}

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();
    }
}

public class ProducerConsumerTest
{

    public static void main(String[] args)
    {
        CubbyHole c = new CubbyHole();
        Producer p1 = new Producer(c, 1);
        Consumer c1 = new Consumer(c, 1);
        p1.start();
        c1.start();
    }
}
```

Semáforos

Un semárofo 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.

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.

Ejemplo del ascensor

```
import java.util.concurrent.Semaphore;
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, quiza sea interrumpido (usando el método interrupt de la clase Thread), y debemos capturar la excepción, en este caso, implicando que se rompió la espera.