1. Comunicación entre hilos en Java: métodos wait(), notify(), notifyAll(), join()

Considera la siguiente situación. Un hilo llamado H se está ejecutando dentro de un método synchronized y necesita acceso a un objeto O que no está disponible temporalmente. ¿Qué debería hacer *el hilo H*? Si ingresa alguna forma de bucle de sondeo que espera a que el objeto O está disponible, el hilo acapara el objeto evitando el acceso de otros hilos a él. Esta es una solución menos que óptima porque parcialmente derrota las ventajas de la programación para un entorno multihilo.

Una mejor solución es hacer que el hilo H renuncie temporalmente al control del objeto O, permitiendo que se ejecute otro hilo. Cuando el objeto O está disponible, se puede notificar a H y reanudar la ejecución. Tal enfoque se basa en alguna forma de **comunicación entre hilos** en la que un hilo puede notificar a otro que está bloqueado y recibir una notificación de que puede reanudar la ejecución. Java admite la comunicación entre hilos con los métodos wait(), notify() y notifyAll().

1. Métodos wait(), notify() y notifyAll()

Los **métodos wait(), notify() y notifyAll()** son parte de todos los objetos porque están implementados por la clase Object. Estos métodos solo deben invocarse desde un contexto sincronizado. Aquí se muestra como se usan.

Cuando un hilo H se bloquea temporalmente para ejecutarse, ocasiona que el hilo quede en reposo y que se libere el monitor para ese objeto O, permitiendo que otro hilo use el objeto O. En un momento posterior, el hilo H en reposo se activa cuando otro hilo entra al mismo monitor y llama a objeto O.

A continuación se muestran las diversas formas de **wait()** definidas por **Object**:

```
final void wait( ) throws InterruptedException

final void wait(long millis) throws InterruptedException

final void wait(long millis, int nanos) throws InterruptedException
```

La primera forma espera hasta ser notificado. La segunda forma espera hasta que se lo notifique o hasta que expire el período especificado de milisegundos. La tercera forma le permite especificar el período de espera en términos de nanosegundos.

ⓘ

Aquí están las formas generales para **notify()** y **notifyAll()**:

```
final void notify()
```

```
final void notifyAll()
```

Una llamada a *notify()* reanuda un hilo de espera. Una llamada a *notifyAll()* notifica a todos los hilos, con el hilo de mayor prioridad ganando acceso al objeto.

Antes de mirar un ejemplo que usa *wait()*, se necesita hacer un punto importante. Aunque *wait()* normalmente espera hasta que se llame a *notify()* o *notifyAll()*, existe la posibilidad de que, en casos muy raros, el hilo de espera se pueda activar debido a una *falsa alarma*.

Las condiciones que conducen a una activación falsa son complejas. Sin embargo, Oracle recomienda que, debido a la remota posibilidad de una activación falsa, las llamadas a wait() se realicen dentro de un bucle que verifique la condición en la que el hilo está esperando. El siguiente ejemplo muestra esta técnica.

2. Ejemplo del uso de wait() y notify()

Para comprender la necesidad y la aplicación de **wait()** y **notify()**, usaremos un programa que simula el tic-tac de un reloj mostrando las palabras *Tic* y *Tac* en la pantalla.

Para lograr esto, crearemos una clase llamada TicTac que contiene dos métodos: *tic()* y *tac()*. El método *tic()* muestra la palabra “Tic”, y *tac()* muestra “Tac”. Para ejecutar el reloj, se crean dos hilos, uno que llama a *tic()* y otro que llama a *tac()*. El objetivo es hacer que los dos hilos se ejecuten de forma tal que la salida del programa muestre un “Tic Tac” consistente, es decir, un patrón repetido de un *tic* seguido de un *tac*.

```
//Uso de wait() y notify() para crear un reloj que haga tictac.

class TicTac{
    String estado;

    synchronized void tic(boolean corriendo){
        if (!corriendo){
            estado="ticmarcado";
            notify();
            return;
        }
        System.out.print("Tic ");
        estado="ticmarcado";
        notify();

        try {
            while (!estado.equals("tacmarcado"))
                wait();
        }catch (InterruptedException exc){
            System.out.println("Hilo interrumpido.");
        }
    }

    synchronized void tac(boolean corriendo){
        if (!corriendo){
            estado="tacmarcado";
            notify();
            return;
        }
        System.out.println("Tac");
        estado="tacmarcado";
        notify();

        try {
            while (!estado.equals("ticmarcado"))
                wait();
        }catch (InterruptedException exc){
            System.out.println("Hilo interrumpido.");
```

```
        }
    }
}

class MiNHilo implements Runnable{
    Thread hilo;
    TicTac ttob;

    MiNHilo(String nombre, TicTac tt){
        hilo=new Thread(this,nombre);
        ttob=tt;
    }

    public static MiNHilo crearEIniciar(String nombre, TicTac tt){
        MiNHilo miNHilo=new MiNHilo(nombre,tt);
        miNHilo.hilo.start(); //Inicia el hilo
        return miNHilo;
    }
    public void run(){
        if (hilo.getName().compareTo("Tic")==0){
            for (int i=0; i<5; i++) ttob.tic(true);
            ttob.tic(false);
        }else {
        for (int i=0; i<5;i++) ttob.tac(true);
        ttob.tac(false);
        }
    }
}
class ComHilos {
    public static void main(String[] args) {
        TicTac tt=new TicTac();
        MiNHilo mh1=MiNHilo.crearEIniciar("Tic",tt);
        MiNHilo mh2=MiNHilo.crearEIniciar("Tac",tt);

        try {
            mh1.hilo.join();
            mh2.hilo.join();
        }catch (InterruptedException exc){
            System.out.println("Hilo principal interrumpido.");
        }
    }
}
```

Aquí está la salida producida por el programa::

```
Tic Tac
Tic Tac
Tic Tac
Tic Tac
Tic Tac
```

3. Explicación del código

Echemos un vistazo de cerca a este programa. El corazón del reloj es la clase *TicTac*. Contiene dos métodos, *tic()* y *tac()*, que se **comunican entre sí** para garantizar que un ***Tic*** siempre va seguido de un ***Tac***, que siempre va seguido de un *Tic*, y así sucesivamente. Observe el campo de estado. Cuando el reloj se está ejecutando, el estado mantendrá la cadena *"ticmarcado*" o "*tacmarcado*", que indica el estado actual del reloj. En *main()*, se crea un objeto *TicTac* llamado *tt*, y este objeto se usa para iniciar dos hilos de ejecución.

Los hilos se basan en objetos de tipo ***MiNHilo***. Tanto el constructor *MiNHilo* como el método ***crearEIniciar()*** tienen dos argumentos. El primero se convierte en el nombre del hilo. Esto será "Tic" o "Tac". El segundo es una referencia al objeto *TicTac*, que es *tt* en este caso. Dentro del método ***run()*** de *MiNHilo*, si el nombre del hilo es "Tic", se realizan llamadas a *tic()*. Si el nombre del hilo es "Tac", se llama al método *tac()*. Se hacen cinco llamadas que pasan "true" como un argumento a cada método. El reloj funciona mientras se pase ***true***. Una llamada final que pasa ***false*** a cada método detiene el reloj.

El método *tac()* es una copia exacta de *tic()* excepto que muestra "Tac" y establece el estado en "tacmarcado". Por lo tanto, cuando se ingresa, muestra "Tac", llama a *notify()* y luego espera. Cuando se ve como una pareja, una llamada a *tic()* solo puede ser seguida por una llamada a *tac()*, que solo puede ser seguido por una llamada a *tic()*, y así sucesivamente. Por lo tanto, los dos métodos se sincronizan mutuamente.

El motivo de la llamada a *notify()* cuando se detiene el reloj es permitir que una llamada final a *wait()* tenga éxito. Recuerde, tanto *tic()* como *tac()*, ejecutan una llamada a *wait()* luego de mostrar su mensaje. El problema es que cuando se detiene el reloj, uno de los métodos seguirá esperando. Por lo tanto, se requiere una llamada final a *notify()* para que se ejecute el método de espera.

Como experimento, eliminamos esta llamada a *notify()* y ver qué sucede. Como verá, el programa se "colgará" y deberá presionar *CTRL-C* para salir. La razón de esto es que cuando la llamada final a *tac()*, llama a *wait()*, no hay una llamada correspondiente a *notify()* que permita concluir *tac()*. Por lo tanto, *tac()* simplemente se queda allí, esperando por siempre.

Antes de continuar, si tiene alguna duda de que las llamadas a ***wait()*** y ***notify()*** son realmente necesarias para que el "reloj" funcione correctamente, sustituya esta versión de *TicTac* en el programa anterior. Tiene todas las llamadas a *wait()* y *notify()* eliminadas.