

Programación multihilo III

Programación de servicios y procesos

Contenidos:

- 1) Mecanismos de comunicación y sincronización de hilos. Paso de mensajes

Sincronización de hilos. wait, notify y notifyAll

Ya hemos visto un tipo de problema en el que varios hilos comparten recursos y se sincroniza el acceso a estos recursos mediante el uso de monitores. Hasta ahora, una vez que un hilo obtiene el bloqueo de un monitor, puede hacer uso del mismo de forma indiscriminada, sin tener en cuenta ninguna otra condición.

Ahora vamos a ver cómo, en función del estado de los recursos, cada uno de los hilos podrá realizar determinadas acciones o no, permitiendo que los hilos se queden a la espera de un cambio de estado que podrá ser notificado por otros hilos. Para ello, además de un mecanismo de bloqueo sobre los recursos compartidos, será necesario un mecanismo de espera para que, en el caso de que el estado de los recursos compartidos no permita a un hilo realizar una acción, la ejecución del hilo quede en suspenso a la espera de que esa condición se cumpla.

El mecanismo es de **espera no activa**, es decir, no se debe consumir tiempo del procesador ni recursos del sistema para comprobar si es posible continuar con la ejecución, mientras no se reciba una notificación de que el estado ha cambiado y podría permitir que el hilo continúe su ejecución.

Sincronización de hilos. wait, notify y notifyAll

Para resolver esta situación vamos a utilizar los métodos wait, notify y notifyAll (son propios de la clase Object, por lo que todas las clases en Java disponen de ellos):

- ❑ **wait():** un hilo que llama al método wait() de un cierto objeto queda suspendido hasta que otro hilo llame al método notify() o notifyAll() del mismo objeto.
- ❑ **wait(long tiempo):** Como el caso anterior, solo que ahora el hilo también puede reanudarse (pasar a "ejecutable") si ha concluido el tiempo pasado como parámetro.
- ❑ **notify():** despierta sólo a uno de los hilos que realizó una llamada a wait(), notificándole que ha habido un cambio de estado sobre el objeto. Si varios hilos están esperando el objeto, solo uno de ellos es elegido para ser despertado. La elección es arbitraria.
- ❑ **notifyAll():** despierta todos los hilos que están esperando el objeto.

Todos estos métodos se tienen que invocar desde segmentos de código de un hilo que disponga de un monitor, como, por ejemplo, un bloque o segmento sincronizados, y obliga a capturar una excepción del tipo *InterruptedException*.

Sincronización de hilos. wait, notify y notifyAll

CONDICIONES

A veces el hilo que se encuentra dentro de una sección crítica no puede continuar, al no cumplirse una condición. Sin embargo, esta condición solo puede ser cambiada por otro hilo desde dentro de su correspondiente sección crítica.

Es necesario que el hilo que no puede continuar libere temporalmente el cerrojo que protege la sección crítica mientras espera a que alguien le avise de la ocurrencia de dicha condición.

Para implementar condiciones se utiliza:

- **Wait:** Libera automáticamente el cerrojo sobre la sección crítica que se está ejecutando.
- **Notify:** Implementa wait. Avisa de la ocurrencia de la condición por la que otro hilo espera.

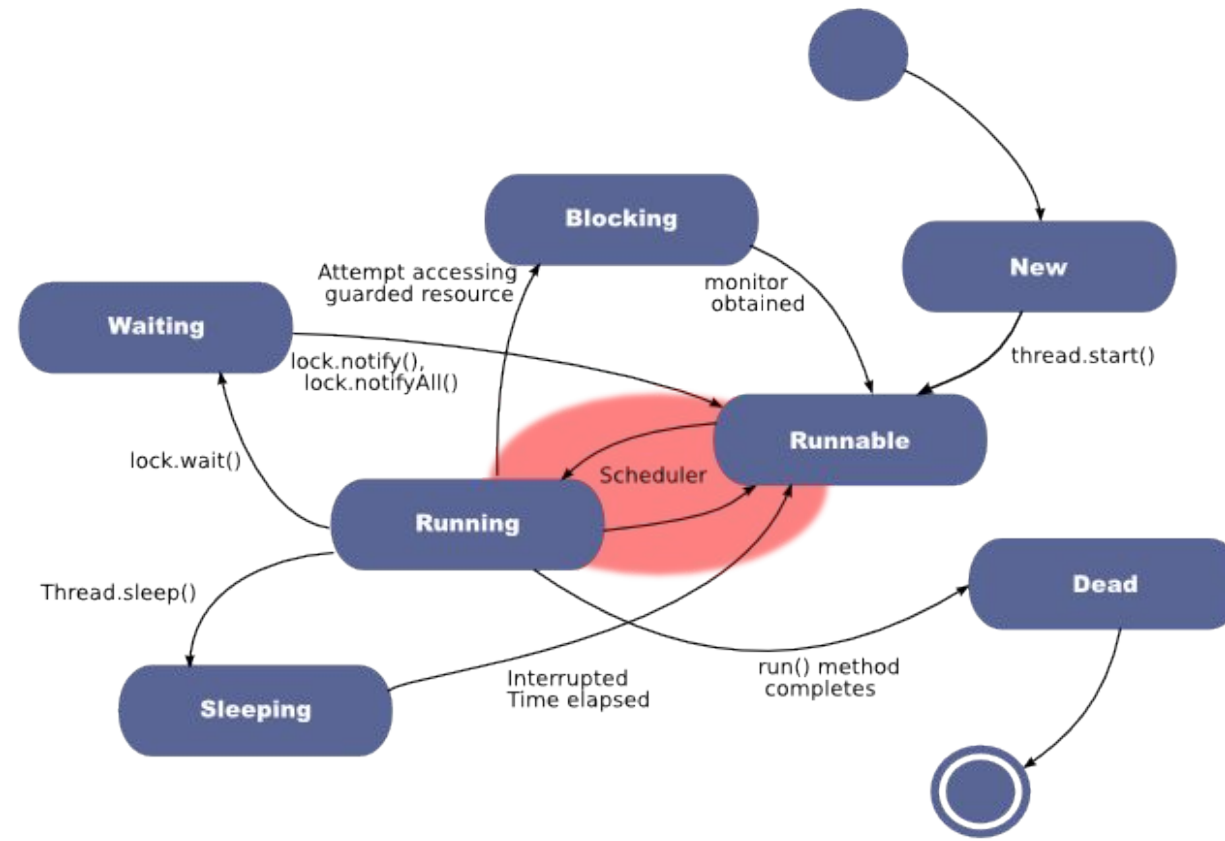
Sincronización de hilos. wait, notify y notifyAll



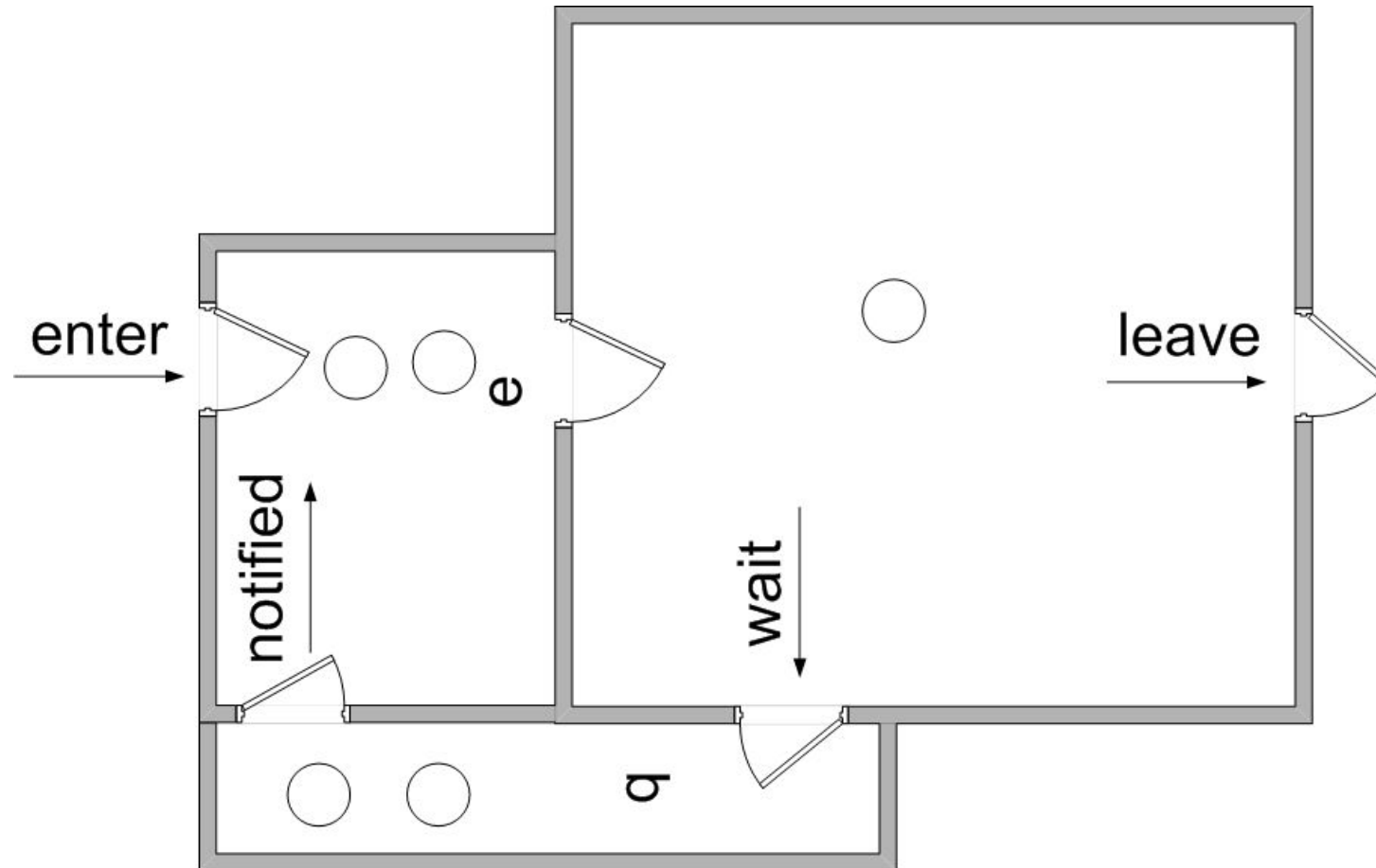
Contexto de ejecución de los métodos de sincronización

These methods need to be called from synchronized context, otherwise it will throw `java.lang.IllegalMonitorStateException`.

Sincronización de hilos. wait, notify y notifyAll



Sincronización de hilos. wait, notify y notifyAll



Sincronización de hilos. wait, notify y notifyAll

Veamos un ejemplo con el código de las clases del proyecto UT3_EjemploLibro

- **Clase Book:** Una clase sencilla que representa un libro con su título, una variable booleana que le diga si el libro está acabado y métodos para que el escritor de por completado el libro.
- **Clase BookWriter:** El escritor. Implementa la interfaz Runnable para poder comportarse como un hilo y en el run empieza a escribir el libro (cuyo título se pasa en el constructor), espera 5 segundos y da por acabado el libro. Avisa a 1 hilo de los que pueden estar esperando.
- **Clase BookReader:** El lector. Implementa la interfaz Runnable para poder comportarse como un hilo y en el run intenta leer el libro que le pasan si está acabado. Si no está acabado, se queda esperando a que le digan que ya puede leerlo.
- **Principal:** instancia 2 lectores, Rosa y Juan, un libro (El Quijote) y un escritor. Los 2 lectores intentan leer antes de que el escritor escriba el libro, se quedan esperando y finalmente uno acaba leyendo.

Sincronización de hilos. wait, notify y notifyAll



¿notify() o notifyAll()?

Todo dependerá del sistema que estemos programando, pero por norma general, si queremos que tras modificar el estado del sistema sólo continúe un hilo, llamaremos a notify().

Sino, debería utilizarse notifyAll(). Si todo está bien programado el hilo comprobará si puede seguir y, en caso contrario, volverá a hacer un wait() y seguir esperando, por eso no supone un problema que que más de un hilo se active.

El uso de notify() supone un mayor riesgo de que se produzcan bloqueos indefinidos de hilos a la espera de notificaciones que nunca van a llegar, siendo este bloqueo diferente de un interbloqueo o deadlock. Debemos ser muy cuidadosos con la programación de los mecanismos de sincronización.

Hay que tener en cuenta también que debería haber al menos una llamada notify() por cada wait() que se haya realizado, aunque eso tampoco asegura que algún hilo no se quede bloqueado.

Modelo productor-consumidor

Un problema típico de sincronización es el que representa el modelo Productor-Consumidor. Se produce cuando uno o más hilos producen datos a procesar y otros hilos los consumen. El problema surge cuando el productor produce datos más rápido que el consumidor los consuma, dando lugar a que el consumidor se salte algún dato.

Igualmente, el consumidor puede consumir más rápido que el productor produce, entonces el consumidor puede recoger varias veces el mismo dato o puede no tener datos que recoger o puede detenerse, etc.

- ▣ **El problema del Productor-Consumidor:** Modela el acceso simultáneo de varios hilos a una estructura de datos u otro recurso, de manera que unos hilos producen y almacenan los datos en el recurso y otros hilos (consumidores) se encargan de eliminar y procesar esos datos.

Sincronización de hilos. Productor- Consumidor

El problema del productor-consumidor

- Un proceso produce elementos de información.
- Un proceso consume elementos de información.
- Se tiene un espacio de almacenamiento intermedio.



Sincronización de hilos. Productor- Consumidor

Si no se implementasen medidas de control, ya hemos visto que podrían darse situaciones anómalas como:

- El consumidor puede obtener los elementos producidos más de una vez, excediendo la producción del mismo (poder dejar la cuenta en números rojos en el ejemplo del banco, o que un lector pueda leer un libro antes de estar terminado).
- El productor sea más rápido que el Consumidor y genere más información de la que el sistema pueda almacenar, o bien parte de la información que genere se pierda sin que un Consumidor la haya recuperado.
- El Consumidor sea más rápido que el Productor y puede terminar consumiendo dos o más veces el mismo valor, generando información inconsistente en el sistema.

Todas estas circunstancias son las que conocemos como condiciones de carrera o race conditions.

Sincronización de hilos. Productor- Consumidor

Diferencia con otros problemas:

□ **Exclusión mutua:**

- ✓ En el caso de la exclusión mutua solamente se permitiría a un proceso acceder a la información.
- ✓ No se permitiría concurrencia entre lectores.

□ **Productor consumidor:**

- ✓ En el productor/consumidor los dos procesos modifican la zona de datos compartida.

Ejemplo Productor - Consumidor

Este modelo se basa en tres clases, aunque dependiendo del problema, podemos encontrarnos que no tenemos productor o consumidor. Vemos un ejemplo:

Se definen 3 clases:

- La clase Cola que será el objeto compartido entre el productor y el consumidor
- La clase productor produce números y los coloca en la cola
- La clase consumidor consume los números y lo saca de la cola.
-

El productor genera números de 0 a 5 en un bucle for, y los pone en el objeto Cola mediante el método put(); después se visualiza y se hace un pausa con sleep(), durante este tiempo el hilo está en el estado Not Runnable (no ejecutable).

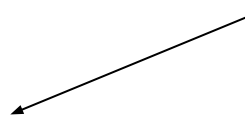
Este ejemplo está en recursos multihilo III: UT3_2_Productor_Consumidor_1

Ejemplo Productor - Consumidor

```
public class Cola {  
    private int numero;  
    private boolean disponible = false; //inicialmente cola vacia  
  
    public int get() {  
        if(disponible) { //hay número en la cola  
            disponible = false; //se pone cola vacía  
            return numero; //se devuelve  
        }  
        return -1; //no hay número disponible, cola vacía  
    }  
  
    public void put(int valor) {  
        numero = valor; //coloca valor en la cola  
        disponible = true; //disponible para consumir, cola llena  
    }  
}
```

La clase cola será la que compartan productor y consumidor.

El productor produce números y los pone en la cola (put), el consumidor los recogerá de la cola (get).



Ejemplo Productor - Consumidor

```
public class Productor extends Thread {
    private Cola cola;
    private int n;

    public Productor(Cola c, int n) {
        cola = c;
        this.n = n;
    }

    public void run() {
        for (int i = 0; i < 5; i++) {
            cola.put(i); //pone el número
            System.out.println(i + "=>Productor : " + n
                               + ", produce: " + i);

            try {
                sleep(100);
            } catch (InterruptedException e) { }
        }
    }
}
```

El productor produce números. Recibe el objeto Cola en el constructor y en su "run" sólo debe generar números.


Tiene un *sleep* de parada en cada vuelta

Ejemplo Productor - Consumidor

```
public class Consumidor extends Thread {  
    private Cola cola;  
    private int n;  
  
    public Consumidor(Cola c, int n) {  
        cola = c;  
        this.n = n;  
    }  
  
    public void run() {  
        int valor = 0;  
        for (int i = 0; i < 5; i++) {  
            valor = cola.get(); //recoge el número  
            System.out.println(i + "=>Consumidor: " + n  
                               + ", consume: " + valor);  
        }  
    }  
}
```

El consumidor recoge los números. No tiene un sleep y por tanto no están sincronizados.

Meterle un sleep algo ayudaría, pero seguiría sin estar "sincronizado".



Ejemplo Productor - Consumidor

```
public class Product_Consum {  
    public static void main(String[] args) {  
        Cola cola = new Cola();  
  
        Productor p = new Productor(cola, 1);  
        Consumidor c = new Consumidor(cola, 1);  
  
        p.start();  
        c.start();  
  
    }  
}
```

La clase con el método main que simplemente instancia las clases y las arranca.

```
<terminated> Product_Consum [Java Applicat  
0=>Productor : 1, produce: 0  
0=>Consumidor: 1, consume: 0  
1=>Consumidor: 1, consume: -1  
2=>Consumidor: 1, consume: -1  
3=>Consumidor: 1, consume: -1  
4=>Consumidor: 1, consume: -1  
1=>Productor : 1, produce: 1  
2=>Productor : 1, produce: 2  
3=>Productor : 1, produce: 3  
4=>Productor : 1, produce: 4
```

Al ejecutar se produce la siguiente salida, en la que se puede observar que el consumidor va más rápido que el productor (al que se le puso un sleep()) y no consume todos los números cuando se producen; el numerito de la izquierda de cada fila representa la iteración:

Ejemplo Productor - Consumidor

Para sincronizar al productor y al consumidor, el primer paso es que los hilos estén sincronizados.

Primero hemos de declarar los métodos `get()` y `put()` de la clase `Cola` como `synchronized`, de esta manera el productor y consumidor no podrán acceder simultáneamente al objeto `Cola` compartido; es decir el productor no puede cambiar el valor de la cola cuando el consumidor esté recogiendo su valor; y el consumidor no puede recoger el valor cuando el productor lo esté cambiando.

```
public synchronized int get() {  
    while (!disponible) {  
        try {  
            wait();  
        } catch (InterruptedException e) { }  
    }  
    //visualize valor  
    disponible = false;  
    notify();  
    return numero;  
}
```

```
public synchronized void put(int valor) {  
    while (disponible){  
        try {  
            wait();  
        } catch (InterruptedException e) { }  
    }  
    numero = valor;  
    disponible = true;  
    //visualiza valor  
    notify();  
}
```

Ejemplo Productor - Consumidor

En segundo lugar, es necesario mantener una coordinación entre el productor y el consumidor de forma que cuando el productor ponga un número en la cola avise al consumidor de que la cola está disponible para recoger su valor; y al revés, cuando el consumidor recoja el valor de la cola debe avisar al productor de que la cola ha quedado vacía. A su vez, el consumidor deberá esperar hasta que la cola se llene y el productor esperará hasta que la cola esté nuevamente vacía para poner otro número.

Para mantener esta coordinación usamos los métodos `wait()`, `notify()` y `notifyAll()`. Estos sólo pueden ser invocados desde dentro de un método sincronizado o dentro de un bloque o una sentencia sincronizada.

Ejemplo Productor - Consumidor

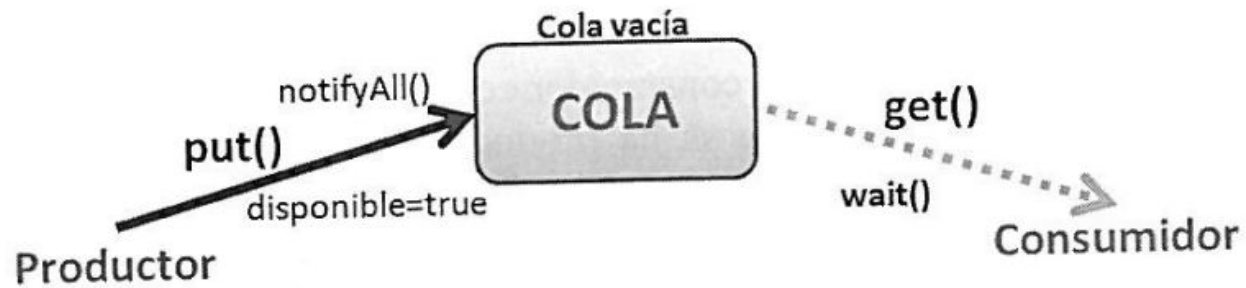


Figura 2.6. Método `get()` espera.

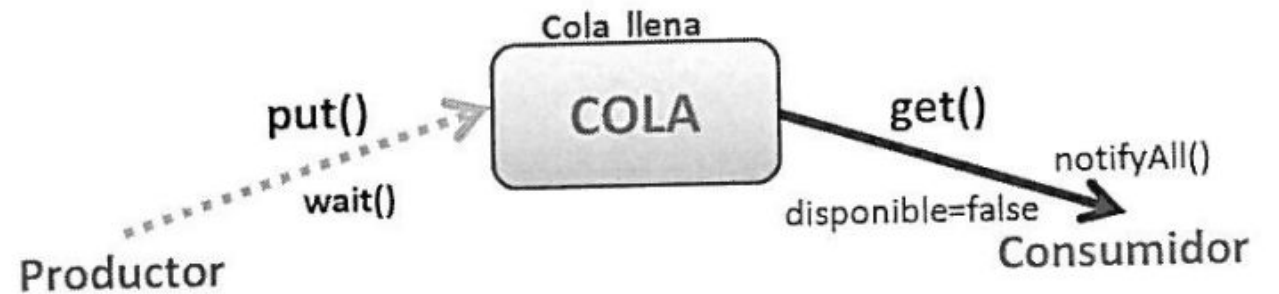


Figura 2.7. Método `get()` devuelve valor, `put()` espera.

Ejemplo Productor - Consumidor

```
public synchronized int get() {  
    while (!disponible) {  
        try {  
            wait();  
        } catch (InterruptedException e) { }  
    }  
    //visualize valor  
    disponible = false;  
    notify();  
    return numero;  
}
```

Si no hay número disponible debemos esperar.

Si he pasado del wait() es porque disponible es true... lo ha debido poner a true el productor y nos lo ha "notificado" para que podamos avanzar

En este caso la cola no tiene "profundidad" y se vacía cada vez que "entrega" un número. Se pone el disponible a false y se le "notifica" al productor.

Ejemplo Productor - Consumidor

```
public synchronized void put(int valor) {  
    while (disponible){  
        try {  
            wait();  
        } catch (InterruptedException e) { }  
    }  
    numero = valor;  
    disponible = true;  
    //visualiza valor  
    notify();  
}
```

La cola no admite profundidad, por lo que si el número está ocupado debemos esperar.

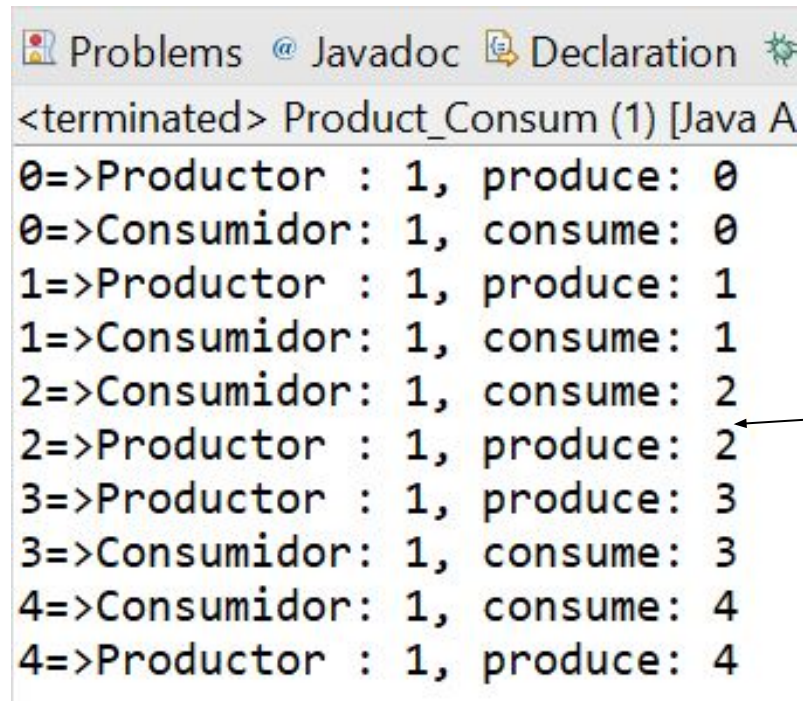
Si he pasado del wait() es porque disponible es false... lo ha debido poner a false el consumidor y nos lo ha “*notificado*” para que podamos avanzar

Asignamos el valor al número, ponemos el disponible a true y se le “*notifica*” al consumidor para que lo pueda entregar.

Ejemplo Productor - Consumidor

El resto de clases, Productor, Consumidor y Product_Consum, no cambian y son exactamente iguales que antes.

Ahora al ejecutar vemos que en cada iteración el productor y el consumidor consumen el mismo número



```
<terminated> Product_Consum (1) [Java A
0=>Productor : 1, produce: 0
0=>Consumidor: 1, consume: 0
1=>Productor : 1, produce: 1
1=>Consumidor: 1, consume: 1
2=>Consumidor: 1, consume: 2
2=>Productor : 1, produce: 2
3=>Productor : 1, produce: 3
3=>Consumidor: 1, consume: 3
4=>Consumidor: 1, consume: 4
4=>Productor : 1, produce: 4
```

Hay que tener cuidado con la salida por pantalla. Todos los threads están usando System.out a la vez y los resultados que se muestran por pantalla, concretamente el orden en el que se muestran, no siempre es el mismo orden en el que se han producido. Por eso es importante que las salidas de los hilos se muevan dentro de los bloques sincronizados.

Si no controlamos la forma de mostrar la salida podemos encontrarnos con problemas que están bien resueltos pero que la salida nos dice lo contrario.

Este ejemplo está en recursos multihilo III: UT3_2_Productor_Consumidor_2

Sincronización de hilos. Productor- Consumidor



Número de hilos por tipo

En el ejemplo se crea un hilo de cada tipo. Esto no tiene porqué ser así.

Cada problema determinará el número de hilos *Productores* y *Consumidores* necesarios, por lo que será en este método *main*, o en algún otro método de la *ClasePrincipal* donde se realice la gestión de los hilos.

De igual forma, dependerá de cada problema si el hilo principal debe esperar a que el resto finalice o no.