2.7.3. BLOQUEO DE HILOS

En el siguiente ejemplo creamos una clase que define un método que recibe un String y lo pinta:

```
class  ObjetoCompartido{
        public  void  PintaCadena(String  s){
                System.out.print(s);
        }
)//  ObjetoCompartido
```

Para usarla definimos un método main() en el que se crea un objeto de esa clase que además será compartido por dos hilos del tipo HiloCadena. Los hilos usarán el método del objeto compartido para pintar una cadena, esta cadena es enviada al crear el hilo (new HiloCadena (objeto compartido, cadena)):

```
public class  BloqueoHilos{
        public  static void main(String[]  args)    {
                ObjetoCompartido  com = new  ObjetoCompartido();
                HiloCadena a =  new  HiloCadena          (com,   "  A  ");
                HiloCadena b  =  new HiloCadena          (com,   "  B  ");
                a.start();
                b.start();
        }
}//BloqueoHilos
```

La clase HiloCadena extiende Thread; en su método run() invoca al método PintaCadena() del objeto compartido dentro de un bucle for:

```
class  HiloCadena extends  Thread  {
        private  ObjetoCompartido  objeto;
        String  cad;

        public  HiloCadena (ObjetoCompartido  c,   String  s)      {
        this.objeto =  c;
        this.cad=s;
        }
public  void run(){
        for(int  j=0;  j<10;  j++)
```

```
                objeto.PintaCadena(cad);
        }//run
}//HiloCadena
```

Se pretende mostrar de forma alternativa los String que inicializa cada hilo y que la salida generada al ejecutar la función main()sea la siguiente: "A B A B A B…".Parece que una primera aproximación para solucionarlo sería sincronizar el trozo de código que hace uso deI objeto compartido (dentro del método run()):

```
synchronized    (objeto){
        for(int  j=0;j<10;  j++)
        objeto.PintaCadena(cad);
}
```

Pero al ejecutarlo, la salida no es la esperada ya que la sincronización evita que dos llamadas a métodos o bloques sincronizados del mismo objeto se mezclen; pero no garantiza el orden de las llamadas; y en este caso nos interesa que las llamadas al método PintaCadena() se realicen de forma alternativa. Se necesita por tanto mantener una cierta coordinación entre los dos hilos, para ello se usan los métodos wait(), notify() y notifyAll():

- Objeto.wait(): un hilo que llama al metodo wait() de un cierto objeto queda suspendido hasta que otro hilo llame al metodo notify() o notifyAll() del mismo objeto.
- Objeto.notify(): despierta sólo a uno de los hilos que realizó una llamada a wait() sobre el mismo objeto notificándole de 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.
- Objeto.notifyAll(): despierta todos los hilos que están esperando el objeto.

En el ejemplo, dentro del bloque sincronizado y después de pintar la cadena se invocará al método notify() del objeto compartido para despertar al hilo que este esperando el objeto (notifyAll() cuando varios hilos esperan el objeto). Inmediatamente después se llama al método wait() del objeto para que el hilo quede suspendido y el que estaba en espera tome el objeto para pintar la cadena; el hilo permanecerá suspendido hasta que se produzca un notify() sobre el objeto. El ultimo notify() es necesario para que los hilos finalicen correctamente y ninguno quede bloqueado:

```
public void run(){
        synchronized    (objeto){
                for(int  j=0;  j<10;  j++){
                        objeto.PintaCadena(cad);
                        objeto.notify(); //aviso  que ya he usado el  objeto
                        try{
```

```
                    objeto.wait();//esperar  a que  llegue un notify
                }catch  (InterruptedException  e){
                    e.printStackTrace();
                }
            }//for
            objeto.notify(); //despertar  a  todos  los  wait  sobre  el  objeto
}//fin bloque synchronized
System.out.print("\n"+cad       +  "  finalizado");
}//run
```

La ejecución del ejemplo muestra la siguiente salida:

```
A  B  A  B  A  B  A  B  A  B  A  B  A  B  A  B  A  B  A  B
B       finalizado
A       finalizado
```

Los métodos notify() y wait() pueden ser invocados solo desde dentro de un método sincronizado o dentro de un bloque o una sentencia sincronizada.

2.7.4. El 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.

Por ejemplo, imaginemos una aplicación donde un hilo (el productor) escribe datos en un fichero mientras que un segundo hilo (el consumidor) lee los datos del mismo fichero; en este caso los hilos comparten un mismo recurso (el fichero) y deben sincronizarse para realizar su tarea correctamente.

EJEMPLO PRODUCTOR-CONSUMIDOR

Se definen 3 clases, la clase Cola que será el objeto compartido entre el productor y el consumidor; y las clases Productor y Consumidor. En el ejemplo el productor produce números y los coloca en una cola, estos serán consumidos por el consumidor. El recurso para compartir es la cola con los números.

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):

```
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 nUmero
                System.out.println(i     + "=>Productor: "       + n
                                         + ", produce: "         + i);
                try{
                        sleep(100);
                }catch  (InterruptedException e) { }
        }
}
}
```

La clase Consumidor es muy similar a la clase Productor, solo que en lugar de poner un numero en el objeto Cola lo recoge llamando al método get(). En este caso no se ha puesto pausa, con esto hacemos que el consumidor sea más rapido que el productor:

```
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 numero
        System.out.println(i     + "=>Consumidor:  "    + n
        + ",  consume:  "        + valor);
                }
        }
}
```

La clase Cola define 2 atributos y dos métodos. En el atributo numero se guarda el número entero y el atributo disponible se utiliza para indicar si hay disponible o no un numero en la cola. El método put() guarda un entero en el atributo número y hace que este esté disponible en la cola para que pueda ser consumido poniendo el valor true en disponible (cola llena). El método get() devuelve el entero de la cola si está disponible (disponible=true) y antes pone la variable a false indicando cola vacía; si el numero no está en la cola (disponible false) devuelve -1;

```
public  class  Cola{
        private  int  numero;
        private  boolean  disponible =  false;//inicialmente  cola  vacía

        public int get()  {
                if(disponible)    {//hay número  en  la  cola
                disponible =  false; //se pone  cola vacia
                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
}
}
```

En el método main() que usa las clases anteriores creamos 3 objetos, un objeto de la clase Cola, un objeto de la clase Productor y otro objeto de la clase Consumidor. Al constructor de las clases Productor y Consumidor le pasamos el objeto compartido de la clase Cola y un numero entero que lo identifique:

```
public  class  Produc_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();
        }
}
```

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 interacción:

```
0=>Productor1 : produce:        0
0=>Consumidor1:  consume:     0
1=>Consumidor1:  consume:     -1
2=>Consumidor1:   consume:    -1
3=>Consumidor1:  consume:     -1
4=>Consumidor1:  consume:     -1
1=>Productor1 : produce:        1
2=>Productor1 : produce:        2
3=>Productor1 : produce:        3
4=>Productor1 : produce:        4
```

En la interacción 0, el productor produce un 0 e inmediatamente el consumidor lo consume, la cola se queda vacía. En la interacción 1 el consumidor consume -1 que indica que la cola esta vacía porque la interacción 1 del productor no se ha producido. En Ia iteración 2 pasa lo mismo el consumidor toma -1 porque el productor aún no ha dejado valor en la cola. Y así sucesivamente. La salida deseada es la siguiente: en cada iteración el productor produce un numero (llena la cola) e inmediatamente el consumidor lo consume (la vacía):

```
0=>Productor1 : produce:        0
0=>Consumidor1:  consume:     0
1=>Productor1 : produce:        1
1=>Consumidor1:  consume:     1
2=>Productor1 : produce:        2
```

2=>Consumidor1: consume: 2

3=>Productor1 : produce: 3

3=>Consumidor1: consume: 3

4=>Productor1 : produce: 4

4=>Consumidor1: consume: 4

ACTIVIDAD 2.7.3

Prueba las clases Productor y Consumidor quitando el método sleep() del productor o añadiendo un sleep al consumidor para hacer que uno sea más rápido que otro. ¿Se obtiene la salida deseada? ¿Qué sería necesario implementar para solventarlo (solo coméntalo, no piques el código)?