CAPÍTULO III: Programación Concurrente en JAVA I: Programación con hilos

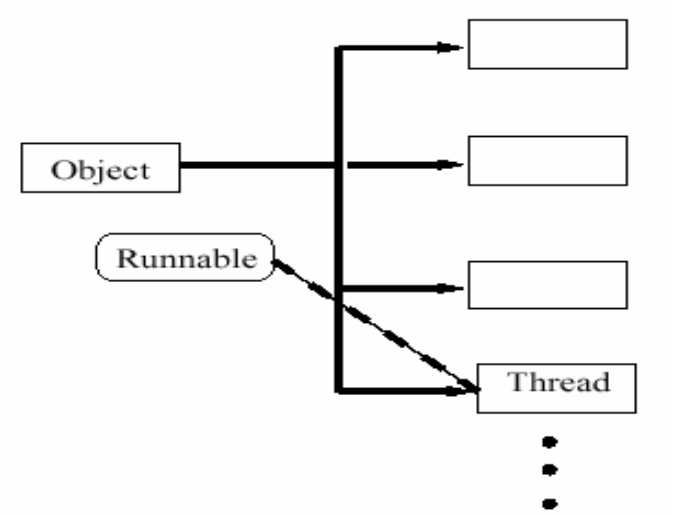
III. PROGRAMACION CON HILOS (THREADS) EN JAVA

Java contiene una serie de clases y construcciones específicas para programar de forma concurrente:

- *java.Lang:Thread*: para el inicio y control de los hilos
- *java.Lang.Object*: para la sincronización de hilos en la compartición de recursos con instrucciones que definen variables de condición:
    a) wait( )
    b) notify( )
    c) notifyAll( )
- synchronized, volatile: Que son palabras reservadas utilizadas en la definición de candados y semáforos para programar la exclusión mutua en la sincronización de los hilos.

- La programación concurrente en Java (con hilos) es fundamental en la implementación de buenos programas:

    a) Permite animaciones de applets
    b) Permite la programación de servidores de red eficientes



- El entorno de programación Java impone a las clases (que definen hilos dentro de aplicaciones software) el implementar la interfaz *Runnable* y a los métodos principales de los hilos el llamarse *run( ).*

III.1. CREACION DE HILOS (THREADS) EN JAVA

Existen dos formas de crear hilos en el lenguaje Java. La primera es utilizar la herencia para extender todas las características de la clase Thread de Java y la segunda es utilizar la implementación de la interfaz *Runnable* en nuestras clases que representarán nuestros hilos.

III.1.1. Heredando de la clase Thread de Java

Extendemos la clase Thread (es decir, heredamos de ella) y redefinimos el método público *run( )* que en dicha clase esta definido como abstracto:

Veamos un sencillo programa de dos threads que imprimen las palabras *"ping" y "pong"* a velocidades diferentes:

```
class PingPong extends Thread
   {
       String palabra;          // Que palabra a imprimir!!!
       int retardo;             // Duración de la pausa

       public PingPong(String palabra, int retardo)
        {
         this.palabra=palabra;
         this.retardo=retardo;
        }

       public void run()
          {
              try {
                   for(;;)
                    {
                       System.out.print(palabra+" ");
                       sleep(retardo);
                   }
                   }catch(InterruptedException e)
                      {
                            return;   // acabar este thread
                       }
          }

       public static void main(String[] args)
          {
              PingPong ping=new PingPong("ping",33);
              PingPong pong=new PingPong("PONG",100);

              ping.start();
              pong.start();
          }
   }
```

Definimos un tipo de hilo llamado *PingPong*. Su método *run( )* itera para siempre, imprimiendo el campo *palabra* y durmiendo durante *retardo* microsegundos. En el método *main( )* se crean dos objetos *PingPong* cada uno de ellos con su propia palabra y ciclo de retardo, e invoca el método *start( )* de cada uno de los objetos thread que es quien ejecuta el método *run( )* lanzando el hilo a ejecución. La salida producida es la que se muestra a continuación:

Un hilo puede tener nombre, ya sea como parámetro *String* para el constructor o como parámetro de una llamada *setName*. Se puede obtener el nombre actual de un thread invocando a *getName( )*. El siguiente código muestra un ejemplo de ello:

```
class A extends Thread
   {
    public A()
      { super(); }

    public A(String nombre)
        { super(nombre); }

    public void run()
       { System.out.println("Nombre= "+getName()); }
   }

class B
  {
   public static void main(String[] args)
      {
       A a=new A("Mi hilo");
       A b=new A();
       b.setName("Otro Hilo");

       a.start();
       b.start();
      }
  }
```

La salida de este ejemplo es:

III.1.2. Implementando la interfaz Runnable de Java

Se implementa la interfaz Runnable en una clase con un método público *run( )*. Veamos una versión *Runnable* de la clase *PingPong* creada en la sección anterior. Si se

comparan ambas versiones se verá que parecen casi idénticas. Las diferencias principales están en la superclase (*Runnable vs Thread*) y en el programa principal:

```
class RunPingPong implements Runnable
   {
       String palabra;          // Que palabra a imprimir!!!
       int retardo;             // Duración de la pausa

       public RunPingPong(String palabra, int retardo)
        {
         this.palabra=palabra;
         this.retardo=retardo;
        }
       public void run()
          {
              try {
                   for(;;)
                     {
                      System.out.print(palabra+" ");
                      Thread.sleep(retardo);
                     }
                   }catch(InterruptedException e)
                       {
                           return;   // acabar este thread
                       }
          }
        public static void main(String[] args)
          {
              Runnable ping=new RunPingPong("ping",33);
              Runnable pong=new RunPingPong("PONG",200);
              new Thread(ping).start();
              new Thread(pong).start();
          }
   }
```

En primer lugar se define una nueva clase que implementa *Runnable*. Su implementación del método *run( )* es la misma que la de *PingPong*. En el *main( )* se crean dos objetos *RunPingPong* con tiempos diferentes; se crea entonces un nuevo objeto *Thread* para cada uno de ellos y se inicia inmediatamente.

Aquí una segunda versión del segundo código fuente mostrado en la sección anterior:

```
class A_version2 implements Runnable
 {
  public void run()
   {
  System.out.println("Nombre="+Thread.currentThread().getName());
   }
 }

class B_version2
  {
   public static void main(String[] args)
      {
       A_version2 a=new A_version2();
       Thread t=new Thread(a,"Mi hilo");
       t.start();
      }
  }
```

Esta segunda forma de crear un hilo implementando *Runnable* es mejor ya que en Java no existe la herencia múltiple. También se puede incluir el hilo dentro del método constructor como lo muestra esta nueva versión del código anterior:

```
class A_version3 implements Runnable
  {
   private Thread t=null;

   public A_version3()
      {
       t=new Thread(this,"Mi hilo");
       t.start();
      }



  public void run()
    {
  System.out.println("Nombre="+Thread.currentThread().getName());
    }

   public static void main(String[] args)
      {
       A_version3 a=new A_version3();
      }
  }
```

III.1.3. Constructores de la clase Thread

Cuatro constructores *Thread* permiten especificar un objeto *Runnable*:

`public Thread(Runnable objeto):` Construye un nuevo Thread que usa el método *run( )* del *objeto* especificado.

`public Thread(Runnable objeto, String nombre):` Construye un nuevo Thread con el *nombre* especificado y usa el método *run( )* del *objeto* especificado.

`public Thread(ThreadGroup grupo, Runnable objeto):` Construye un nuevo Thread en el *ThreadGroup* especificado y usa el método *run( )* del *objeto* especificado.

`public Thread(ThreadGroup grupo, Runnable objeto, String nombre):` Construye un nuevo Thread en el *ThreadGroup* especificado, con el *nombre* especificado y usa el método *run( )* del *objeto* especificado.

III.1.4. Métodos de Instancia más utilizados

- ***start( )*:** Genera un Nuevo Thread de control a partir de los datos del objeto Thread y luego retorna. Invoca el método *run( )* del nuevo thread
- ***join( ):*** Espera para siempre a que acabe el Thread que lo llama, esto es, un Thread espera a que otro termine usando este método.
- ***stop( ):*** Detiene la ejecución de un thread suspendido.

- ***suspend( ):*** Suspende la ejecución de un thread hasta que sea reanudada explícitamente
- ***resume( ):*** Reanuda la ejecución de un thread suspendido.
- ***setPriority(int):*** Asigna una prioridad de ejecución a un Thread
- ***getPriority( ):*** Se obtiene la prioridad de ejecución de un thread corriente.
- ***setDaemon(true):*** Marca un thread como un thread demonio. Un thread demonio es consumable, es decir, cuando acaba el último thread usuario, los posibles threads demonios se detienen y la aplicación esta hecha.
- ***getName( )*:** Obtiene el nombre actual de un Thread

III.1.5. Métodos de clase

Diversos métodos estáticos de la clase *Thread* controlan la planificación del Thread actual.

`public static void sleep(long millis) throws InterruptedException`: Pone el thread actualmente en ejecución a dormir durante al menos el número de milisegundos especificado. "Al menos" significa que no hay garantía de que el thread vaya a despertarse exactamente en el tiempo especificado. La planificación de otro thread puede interferir, así como la resolución y exactitud del reloj del sistema, entre otros factores.

`public static void sleep(long millis, int nanos) throws InterruptedException`: Pone el thread actualmente en ejecución a dormir durante al menos el número de milisegundos y nanosegundos adicionales especificado. Los nanosegundos estan en el rango 0-999999.

`public static void yield()`: Hace que el thread actualmente en ejecución ceda el paso, de modo que puedan ejecutarse otros threads ejecutables. El planificador de los threads elige un thread a ejecutar entre los threads ejecutables. El thread elegido podría ser el que ha cedido el paso, puesto que puede ser el de máxima prioridad entre los threads ejecutables.

El siguiente programa muestra como funciona *yield()*. La aplicación toma una lista de palabras y crea un thread responsable de imprimir cada una de ellas. El primer parámetro para la aplicación dice si cada thread va a ceder el paso después de cada *println( )*, el segundo es el número de veces que debe repetir su palabra cada thread. Los parámetros restantes son las palabras a repetir:

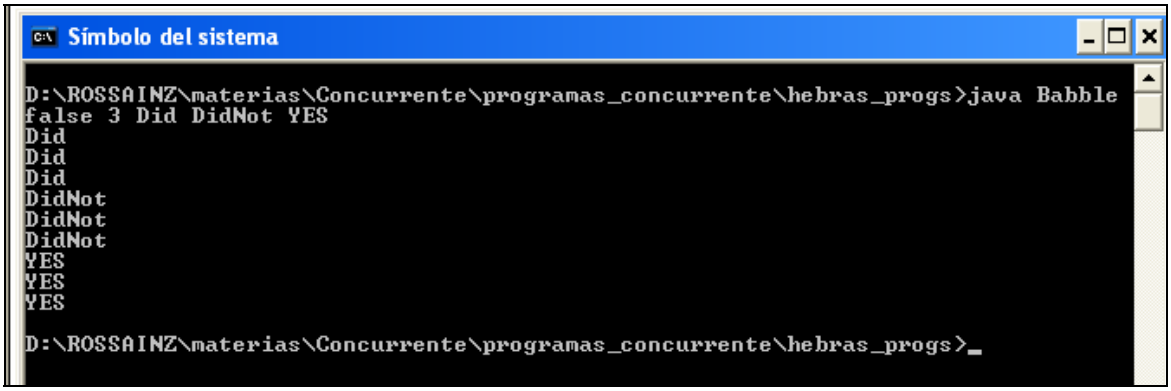
```
public class Babble extends Thread
   {
       static boolean doYield; //¿Ceder paso a otros hilos?
       static int howOften;     // Cuantas veces imprimir
       String word;            // mi palabra

       public Babble(String word)
          {
              this.word=word;
          }
```

```
      public void run()
         {
             for(int i=0; i<howOften;i++)
                {
                    System.out.println(word);
                    if (doYield)
                        yield();   //dar oportunidad a otra palabra
                }
         }

      public static void main(String[] args)
         {
             try{
             howOften=Integer.parseInt(args[1]);
             doYield=new Boolean(args[0]).booleanValue();

             //Crear un thread para cada palabra con prioridad máxima
             //Thread cur=currentThread();
             //cur.setPriority(Thread.MAX_PRIORITY);
             for(int i=2;i<args.length;i++)
                new Babble(args[i]).start();
              }catch(Exception e)
                 {
                  System.out.println("java Babble <true/false> <entero>
                                                      <[palabras ]>");
                  }
         }
   }
```

Cuando los threads no ceden el paso, cada uno de ellos tomará grandes lapsos de tiempo, habitualmente suficiente para terminar todas las impresiones sin que ningún otro thread obtenga ciclos. Así por ejemplo, cuando se ejecuta la aplicación habiendo asignado *false* a *doYield* de la manera siguiente:

```
Babble false 3 Did DidNot Yes
```

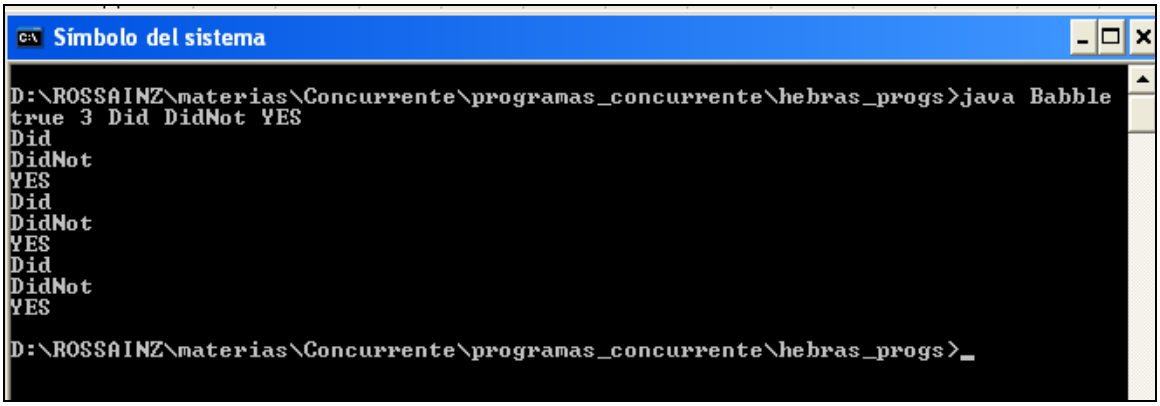
Lo más probable es que la salida sea:



Si cada uno de los threads cede el paso después de cada uno de los *println( )*, tendrán alguna posibilidad de ejecutarse otros threads de impresión. Cuando asignamos *true* a *doYield* con una invocación como la siguiente:

```
Babble true 3 Did DidNot Yes
```

Las cesiones dan a los otros threads una oportunidad de ejecutarse y los otros threads ceden, a su vez, el paso, produciendo una salida que se parece más a:



La salida mostrada es sólo un resultado posible de ejecución de la aplicación. Al volver a ejecutar la aplicación podría darnos un resultado diferente, en otras palabras, la misma implementación daría resultados distintos en diferentes ejecuciones de la aplicación. Al invocar a *yield* da a otros threads posibilidades más equitativas en la obtención de ciclos.

III.2. SINCRONIZACION DE THREADS

Cuando dos cajeros automáticos (threads) necesitan usar el mismo archivo (objeto) existe una posibilidad de entremezclado de operaciones que puede corromper los datos. En el banco, los cajeros *sincronizan* el acceso colocando pequeñas notas en los archivos. El equivalente en *multithreading* es colocar un ***bloqueo*** sobre el objeto. Cuando algún objeto esta bloqueado por algún thread sólo ese thread puede acceder a él.

III.2.1. synchronized

La palabra reservada ***synchronized*** sirve para hacer que un bloque de código o un método sea protegido por un *cerrojo interno de los objetos*. Los threads tienen que adquirir el cerrojo del objeto (obj) para ejecutar el código sincronizado:

```
synchronized(obj)
   {
    // bloque de código sincronizado
   }
```

Si todas las secciones críticas están en el código de un único objeto, podemos utilizar *this* para reverenciarlo:

```
synchronized(this)
   {
    // bloque de código sincronizado
   }
```

El cuerpo entero de un método puede ser código sincronizado:

```
tipo metodo( . . .)
   {
    synchronized(this)
       {
        // código del método sincronizado
       }
   }
```

La siguiente construcción es equivalente a la anterior:

```
synchronized tipo metodo( . . .)
   {
    // código del método sincronizado
   }
```

La sincronización obliga a que la ejecución de dos threads sea *mutuamente excluyente* en el tiempo.

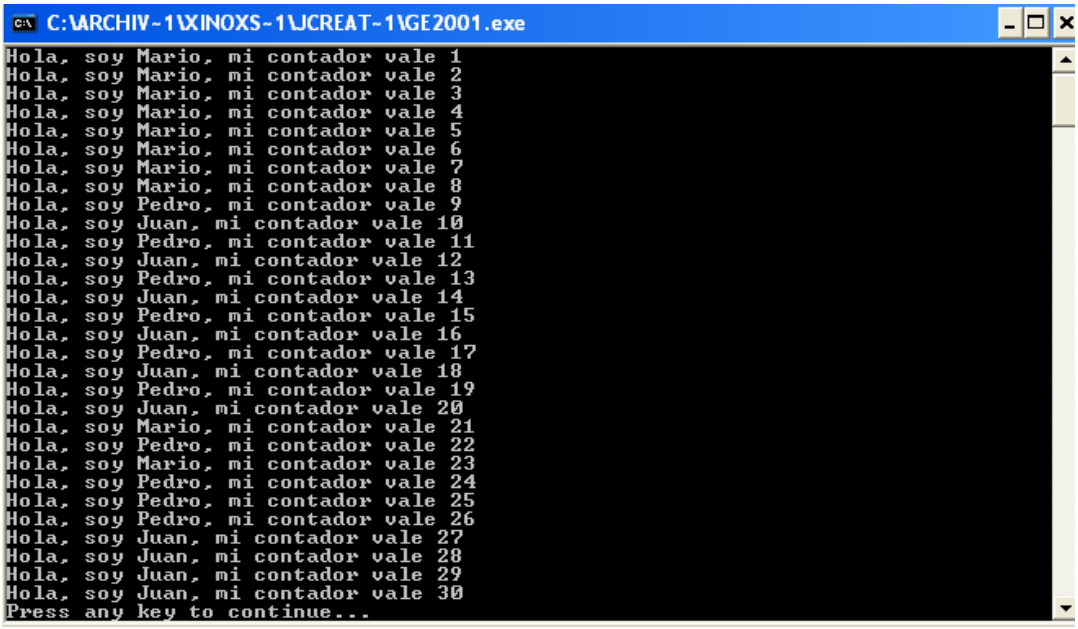
III.2.2. Monitores

La forma de crear un monitor en Java es declarar una clase con sus métodos sincronizados, como por ejemplo: *Un monitor que implementa un contador* cuyo código se muestra a continuación:

```
class Contador                  // MONITOR CONTADOR!!!
    {
       private int actual;

       public Contador(int actual)
          { this.actual=actual; }

       public synchronized void inc()
          { actual++; }

       public synchronized void dec()
          { actual--; }

       public synchronized int valor()
          { return actual; }
    }

class Usuario extends Thread    // Clase hilo usuario!!!
    {
       private Contador cnt;

       public Usuario(String nombre, Contador cnt)
          {
           super(nombre);
           this.cnt=cnt;
          }

       public void run()
          {
           for(int i=0; i<10; i++)
             {
              cnt.inc();
```

```
            System.out.println("Hola, soy "+this.getName()+
                               ", mi Contador vale "+cnt.valor());
          }
        }

    public static void main(String[] args)
       {
        Contador cnt= new Contador(0);
        Usuario u1=new Usuario("Mario",cnt/*new Contador(0)*/);
        Usuario u2=new Usuario("Pedro",cnt/*new Contador(100)*/);
        Usuario u3=new Usuario("Juan",cnt/*new Contador(-20)*/);
        u1.start();
        u2.start();
        u3.start();
       }
  }
```

Donde una corrida de ejecución daría posiblemente el resultado siguiente:



III.2.3. Métodos de espera y notificación

El mecanismo de bloqueo *synchronized* es suficiente para evitar que los threads interfieran entre sí, pero también hace falta una forma de comunicación entre un thread y otro. Con este fín, se define el método ***wait*** para permitir que se espere hasta que se produce una situación y se define ***notify*** para decir a los que esperan que ha ocurrido algo. Esta es la definición de lo que se conoce como ***variable de condición***.

III.2.3.1. Variables de Condición

Los métodos *wait y notify* se definen en la clase *Object* y son heredados por todas las clases. Se aplican a objetos concretos. En un modelo de lectores/escritores, cuando el

lector espera (*wait*) un suceso, está esperando a que algún otro thread le notifique (*notify*) ese suceso sobre el mismo objeto en el que está esperando.

El thread que espera una condición debe hacer algo como lo siguiente:

```
synchronized void doWhenCondition()
   {
    while (!condition)
         wait( );
    //Hacer lo que hay que hacer cuando la condición es cierta
   }
```

Aquí pasan una serie de cosas:

- Todo se ejecuta en un método *sinchronized*. Esto debe ser así, de lo contarior, los contenidos del objeto no son estables. Por ejemplo, si el método no es *synchronized*, después de la sentencia *while* no hay garantía de que la condición sea *trae*, porque algún otro thread puede haber cambiado la situación comprobada por la condición.
- Uno de los aspectos más importantes de la definición de *wait* es que cuando suspende el thread libera *atómicamente* el bloqueo sobre el objeto. Es decir, la suspensión del thread y la liberación del bloqueo ocurren juntos de manera indivisible. De lo contrario, habría un riesgo de carrera: podría producirse un *notify* después de que se liberara el bloqueo pero antes de que se suspendiera el thread. El *notify* no tendría efecto sobre el thread y se perdería. Cuando se reinicia un thread después de haber sido modificado se vuelve a obtener el bloqueo.
- La prueba de la condición debe estar siempre en un bucle. No hay que suponer nunca que ser despertado significa que se ha satisfecho la condición. Dicho de otra forma, no hay que cambiar el *while* por el *if*.

Por otra parte, el método *notify* es invocado por métodos que cambian datos que pueden estar esperando algun otro thread.

```
Synchronized void changeCondition( )
   {
    // Cambiar algún valor usado en una variable de condición
    notify();
   }
```

Pueden estar esperando múltiples threads sobre el mismo objeto. Al usar *notify* se despierta uno de los threads en espera. Como no es posible predecir cuál, habitualmente se despiertan todos los threads en espera usando ***notifyAll***.

El siguiente ejemplo muestra el problema del productor/consumidor analizado en el capítulo II de estas notas en sus tres versiones concurrentes:

```
public class BufferCirc
    {
     private final int BUFSIZE=8;
     private int bufin;
     private int bufout;
     private int[] buffer;

     public BufferCirc()
```

```
        {
         bufin=0;
         bufout=0;
         buffer=new int[BUFSIZE];
        }

     public synchronized int get_item()
        {
         int valor;
         valor=buffer[bufout];
         bufout=(bufout+1)%BUFSIZE;
         return valor;
        }

     public synchronized void put_item(int item)
       {
        buffer[bufin]=item;
        bufin=(bufin+1)%BUFSIZE;
       }
    }
```

III.2.3.1.1. Problema del Productor/Consumidor con buffer circular (versión 1):

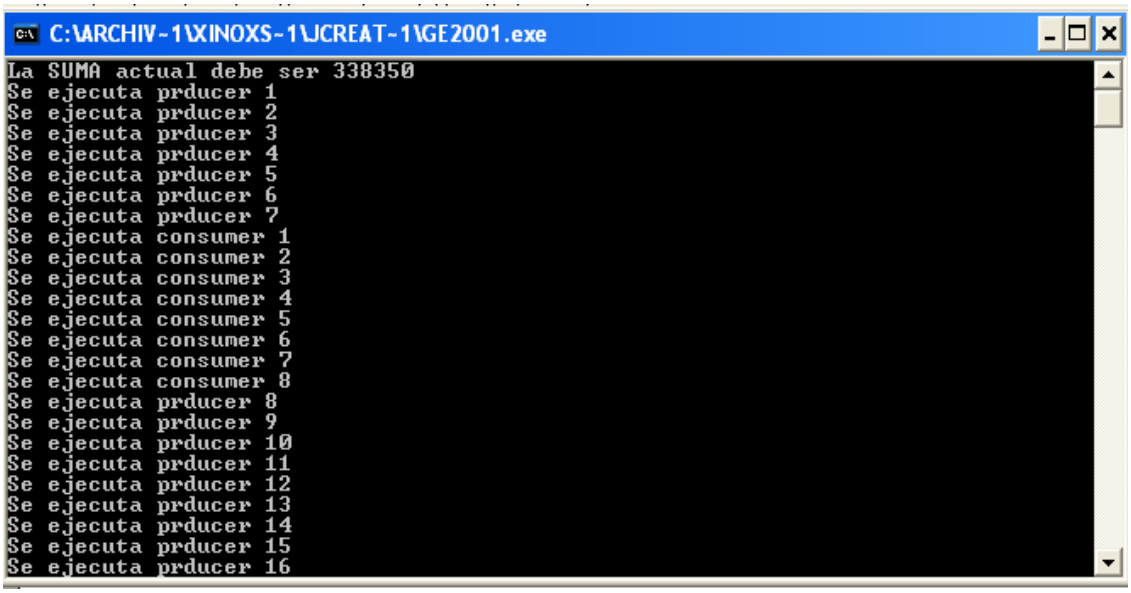
```
public class Productor extends Thread
   {
       private int sumaTam;
       private BufferCirc buf;

    public Productor(int sumaTam, BufferCirc buf)
      {
       this.sumaTam=sumaTam;
       this.buf=buf;
      }

       public void run()
          {
              for(int i=1;i<=sumaTam;i++)
                {
                 System.out.println("Se ejecuta prducer "+i);
                 buf.put_item(i*i);
                }
          }

   }
```

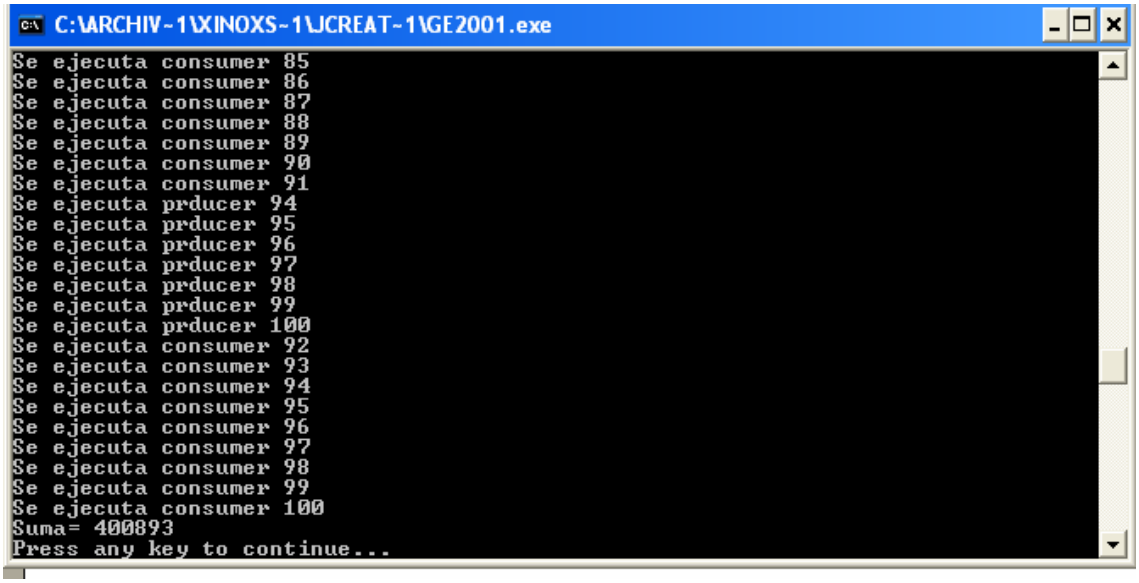
```
public class Consumidor extends Thread
   {
       private int sumaTam;
       private BufferCirc buf;
       public int sum;

    public Consumidor(int sumaTam, BufferCirc buf)
      {
       this.sumaTam=sumaTam;
       this.buf=buf;
       sum=0;
      }
```

```
    public void run()
      {
       for(int i=1;i<=sumaTam;i++)
         {
          System.out.println("Se ejecuta consumer "+i);
          sum+=buf.get_item();
         }
       System.out.println("Suma= "+sum);
      }
   }
```

```
public class ProdCons1
   {
      public static void main(String[] args)
         {
            int total=0;
            for (int i=1;i<=100;i++)
              total+=i*i;
            System.out.println("La SUMA actual debe ser "+total);
            BufferCirc buffer=new BufferCirc();
            Productor p1=new Productor(100,buffer);
            Consumidor c1=new Consumidor(100,buffer);
            p1.start();
            c1.start();

            try{
                  p1.join();
                  c1.join();
               }catch(InterruptedException e){}
         }
   }
```

Salida de ejecución:





III.2.3.1.2. Problema del Productor/Consumidor con buffer circular (versión 2):

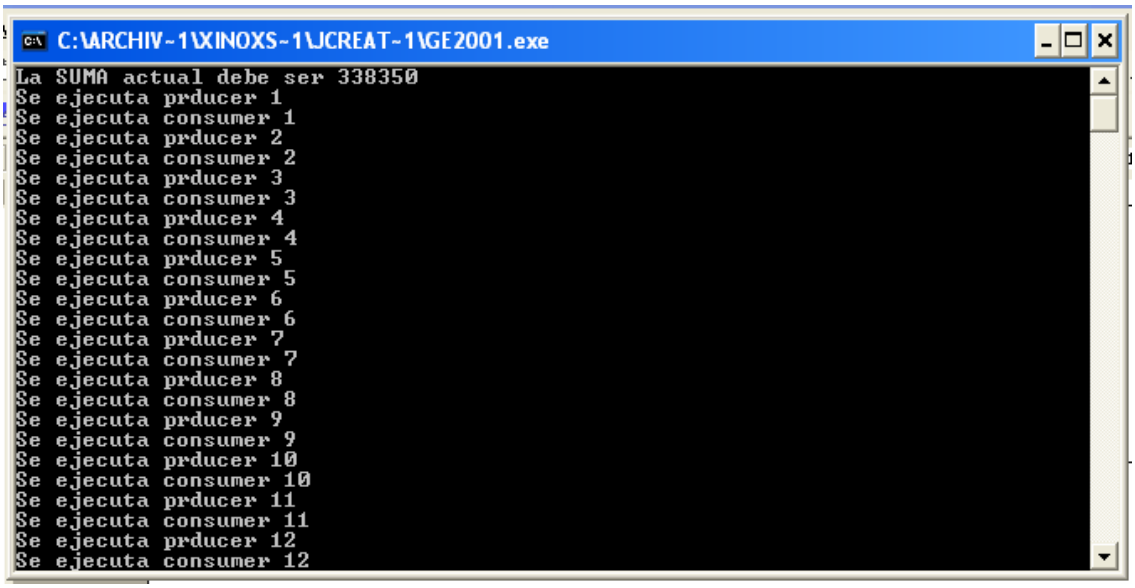
```
public class Productor2 extends Thread
   {
       private int sumaTam;
       private BufferCirc buf;

    public Productor2(int sumaTam, BufferCirc buf)
      {
       this.sumaTam=sumaTam;
       this.buf=buf;
      }

       public void run()
          {
              for(int i=1;i<=sumaTam;i++)
                {
                 System.out.println("Se ejecuta prducer "+i);
                 buf.put_item(i*i);
                 Thread.yield();
                }
          }

   }
```

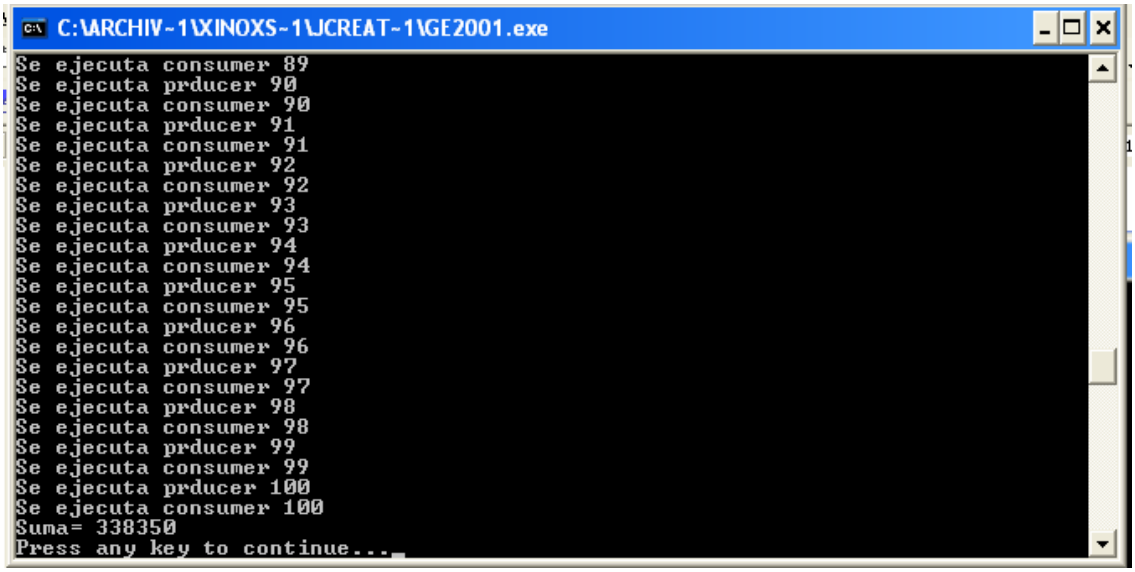
```
public class Consumidor2 extends Thread
   {
       private int sumaTam;
       private BufferCirc buf;
       public int sum;

    public Consumidor2(int sumaTam, BufferCirc buf)
      {
       this.sumaTam=sumaTam;
       this.buf=buf;
       sum=0;
      }
```

```
      public void run()
         {
             for(int i=1;i<=sumaTam;i++)
               {
                System.out.println("Se ejecuta consumer "+i);
                sum+=buf.get_item();
                Thread.yield();
               }
               System.out.println("Suma= "+sum);
         }
   }
```

```
public class ProdCons2
   {
      public static void main(String[] args)
         {
             int total=0;
             for (int i=1;i<=100;i++)
               total+=i*i;
             System.out.println("La SUMA actual debe ser "+total);
             BufferCirc buffer=new BufferCirc();
             Productor2 p1=new Productor2(100,buffer);
             Consumidor2 c1=new Consumidor2(100,buffer);
             p1.start();
             c1.start();
             try{
                    p1.join();
                    c1.join();
                }catch(InterruptedException e){}
         }
   }
```

Salida de Ejecución:





III.2.3.1.3. Problema del Productor/Consumidor con buffer circular (versión 3):

```
public class BufferCirc3
   {
       private final int BUFSIZE=8;
       private int bufin;
       private int bufout;
       private int[] buffer;
       public int nslots;
       public int nitems;
       public int producer_done;

       public BufferCirc3()
          {
              bufin=0;
              bufout=0;
              buffer=new int[BUFSIZE];
              nslots=8;
              nitems=0;
              producer_done=0;
          }

       public synchronized void wait_slot_lock()
          {
              try{
             while (nslots<=0)
              wait();
            }catch(InterruptedException e){}
         nslots--;
          }

       public synchronized boolean wait_item_lock()
          {
              try{
               while((nitems<=0)&&(producer_done==0))
                    wait();
              }catch(InterruptedException e){}
           if ((nitems<=0)&&(producer_done==1))
              return true;
           nitems--;
```

```
          return false;
         }

      public synchronized void notify_slot_lock()
        {
         nslots++;
         notify();
        }

      public synchronized void notify_item_lock()
        {
         nitems++;
       notify();
        }

      public synchronized void put_done()
        {
         producer_done=1;
         notify();
        }

      public synchronized int get_item()
        {
         int valor;
         valor=buffer[bufout];
         bufout=(bufout+1)%BUFSIZE;
         return valor;
        }

    public synchronized void put_item(int item)
      {
       buffer[bufin]=item;
       bufin=(bufin+1)%BUFSIZE;
      }
   }
```

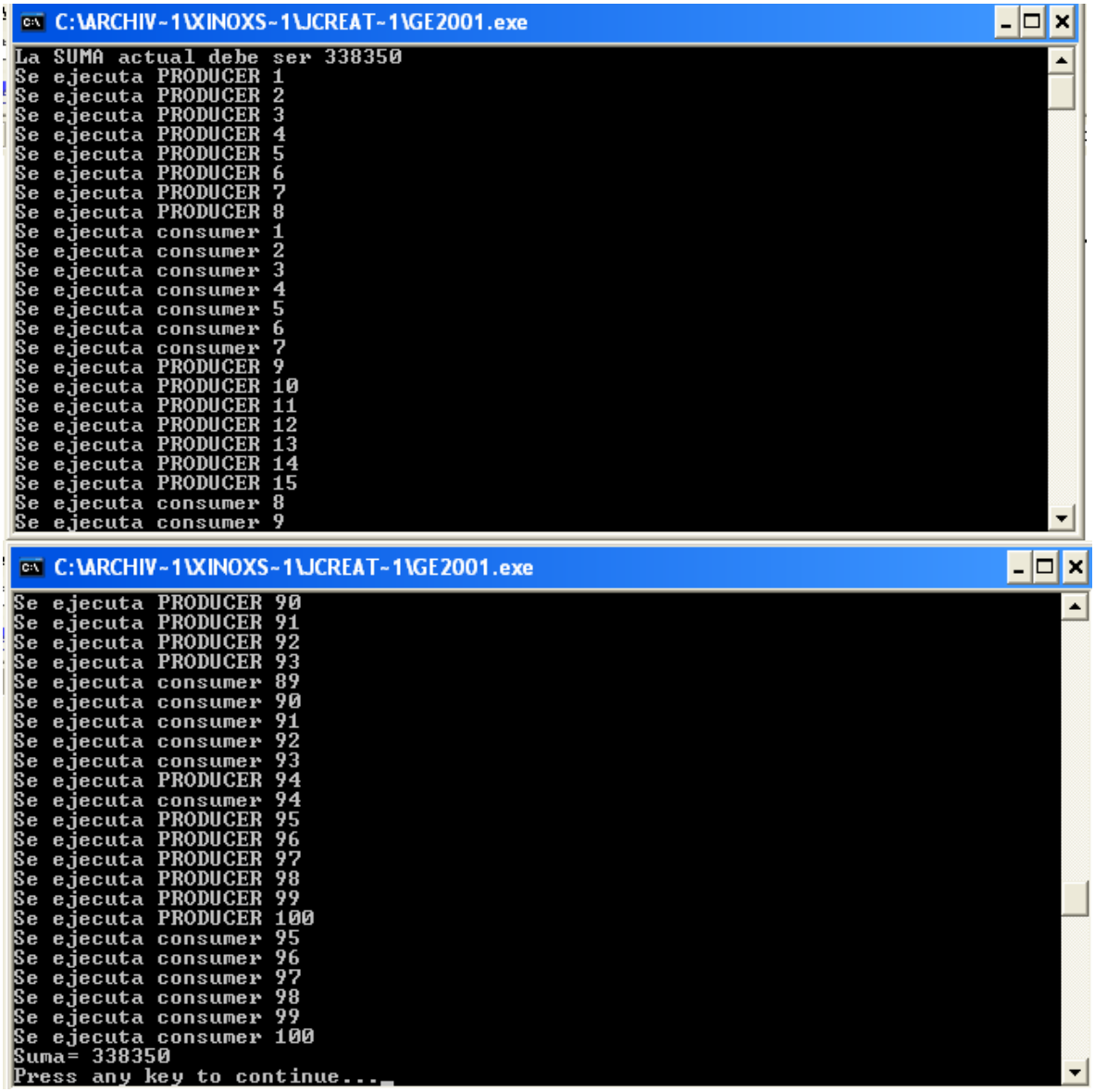
```
public class Productor3 extends Thread
   {
       private int sumaTam;
       private BufferCirc3 buf;

    public Productor3(int sumaTam, BufferCirc3 buf)
      {
       this.sumaTam=sumaTam;
       this.buf=buf;
      }

       public void run()
          {
              for(int i=1;i<=sumaTam;i++)
                {
                 buf.wait_slot_lock();
                 System.out.println("Se ejecuta PRODUCER "+i);
                 buf.put_item(i*i);
                 buf.notify_item_lock();
                }
           buf.put_done();
          }
   }
```

```
public class Consumidor3 extends Thread
```

```
   {
       private int sumaTam;
       private BufferCirc3 buf;
       public int sum;

    public Consumidor3(int sumaTam, BufferCirc3 buf)
      {
       this.sumaTam=sumaTam;
       this.buf=buf;
       sum=0;
      }

       public void run()
          {
              for(int i=1;i<=sumaTam;i++)
                {
                 boolean band=buf.wait_item_lock();
                 if (band)
                    break;
                 System.out.println("Se ejecuta consumer "+i);
                   sum+=buf.get_item();
                 buf.notify_slot_lock();
                }
                System.out.println("Suma= "+sum);
          }
   }
```

```
public class ProdCons3
   {
       public static void main(String[] args)
          {
              int total=0;
              for (int i=1;i<=100;i++)
                 total+=i*i;
              System.out.println("La SUMA actual debe ser "+total);

              BufferCirc3 buffer=new BufferCirc3();
              Productor3 p1=new Productor3(100,buffer);
              Consumidor3 c1=new Consumidor3(100,buffer);
              p1.start();
              c1.start();
              try{
                  p1.join();
                  c1.join();
               } catch(InterruptedException e){}
          }
   }
```

salida de ejecución:



III.2.3.2. wait y notify

Hay tres formas de *wait* y dos de *notify*. Todas ellas son métodos de la clase *Object* y operan sobre el thread actual.

- **`public final void wait(long timeout)throws InterruptedException:`** El thread actual espera hasta que es notificado o hasta que expira el lapso de tiempo (*timeout*) especificado en milisegundos. Si es cero, no habrá lapso de tiempo para *wait* sino que seguirá hasta la notificación.

- **`public final void wait(long timeout, int nanos)throws InterruptedException:`** Un *wait* de resolución más alta, en el que el intervalo de tiempo es la suma de dos parámetros: *timeout* en milisegundos y *nanos* (en nanosegundos en el rango 0-999999).

- **`public final void wait( )throws InterruptedException:`** Equivale a *wait(0)*.

- **`public final void notify():`** Notifica como máximo a un thread a la espera de que cambia una condición.

- **`public final void notifyAll():`** Notifica a todos los threads que estan a la espera de que cambie una condición.

Estos métodos se implementan en *Object*. Sin embargo, pueden ser invocados sólo desde dentro de código *synchronized*, usando el bloqueo para el objeto sobre el que son invocados. La invocación puede ser directa, desde el código *sincrhonized* o indirecta, desde un método invocado en ese código. Si se intenta invocar estos métodos sobre objetos situados fuera del código *synchronized* que ha obtenido el bloqueo se lanzará una *IllegalMonitorStateException*.

III.3. ESTADOS DE UN HILO

Un hilo Java o thread puede estar en alguno de los siguientes estados:

- **`Nuevo`** cuando el objeto hilo es creado: `Thread t= new Thread(. . .);`

- **`Ejecutable`** cuando se invoca al método *start( )* de un hilo: `t.start( );`

- **`Ejecutándose`** cuando el hilo esta ejecutándose en algún CPU de una computadora. Si un hilo en este estado ejecuta el método *yield( )*, deja el procesador y pasa al estado ejecutable: `Thread.yield(); //yield es un método static de Thread`

- **`Suspendido`** cuando un hilo en estado ejecutable o ejecutándose invoca su método *suspend*: t.suspend( ). El hilo vuelve a su estado ejecutable cuando su método *resume()* es invocado por otro hilo: `t.resume();`

- **`Bloqueado`** cuando llama al método *sleep( )*, volviendo al estado ejecutable cuando retorne la llamada: `Thread.sleep(tiempo_milisegundos); // método estático`. Cuando llama a *wait( )* dentro de un bloque o un método sincronizado, volviendo al estado ejecutable cuando otro hilo llame a *notify o a notifyAll*. Cuando llama a *join( )* para sincronizarse con la terminación de otro hilo que aun no ha terminado: `t.join()`. Cuando ejecuta alguna operación de entrada/salida bloqueante.

- **`Bloqueado-Suspendido`** cuando el hilo, estando bloqueada, es suspendida por otro hilo. Si la operación bloqueante termina, el hilo pasa al estado suspendido. Si, aún estando bloqueada, otro hilo llama a su método *resume( )*, el hilo vuelve al estado bloqueado.

- **`Muerto`** cuando su método *run( )* termina o si su método *stop( )* es invocado: `t.stop()`.

III.4. PRIORIDADES Y PLANIFICACION DE THREADS

Cada hilo tiene una prioridad que afecta a su planificación. La prioridad de un thread es inicialmente la misma que la del thread que lo creo. La prioridad se puede cambiar usando ***setPriority( )*** con un valor entero situado entre las constantes de ***Thread.MIN_PRIORITY y MAX_PRIORITY.***

- La prioridad estandar de threads por defecto es ***Thread.NORM_PRIORITY***.

- La prioridad de un thread en ejecución se puede cambiar en cualquier momento.

- El método ***getPriority( )*** devuelve la prioridad de un thread.

- Por tanto, la obtención y cambio de prioridad actual de un hilo se llevan a cabo usando *getPriority() y setPriority().* Ejemplo:

```
hilo = new Thread(objeto);
hilo.setPriority(hilo.getPriority()-1);
hilo.start();
```

- Existen diversas situaciones que provocan la planificación de un nuevo hilo. Cuando el hilo actual:
  - Termina
  - Se bloquea
  - Se para (sleep())
  - Se suspende
  - Ejecuta yield()
  - Otro hilo de mayor prioridad pasa al estado ejecutable

La planificación de un thread actual se controla con los métodos de clase ya mencionados: *sleep( )* e *yield().*