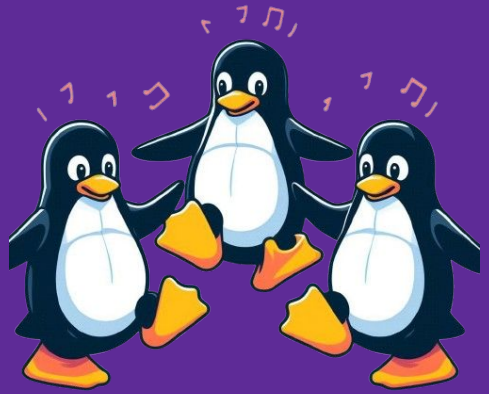
Programación de Procesos y Servicios

UT2.2: Sincronización de Hilos
2º DAM

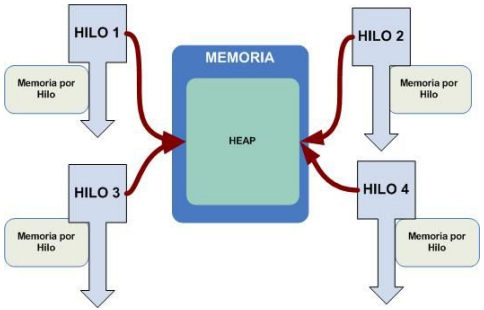


Sincronización de Hilos

Los *threads* se comunican principalmente mediante el intercambio de información a través de variables y objetos en memoria.

- Como todos los *threads* pertenecen al mismo proceso, pueden acceder a toda la memoria asignada a dicho proceso y utilizar las variables y objetos del mismo para compartir información.

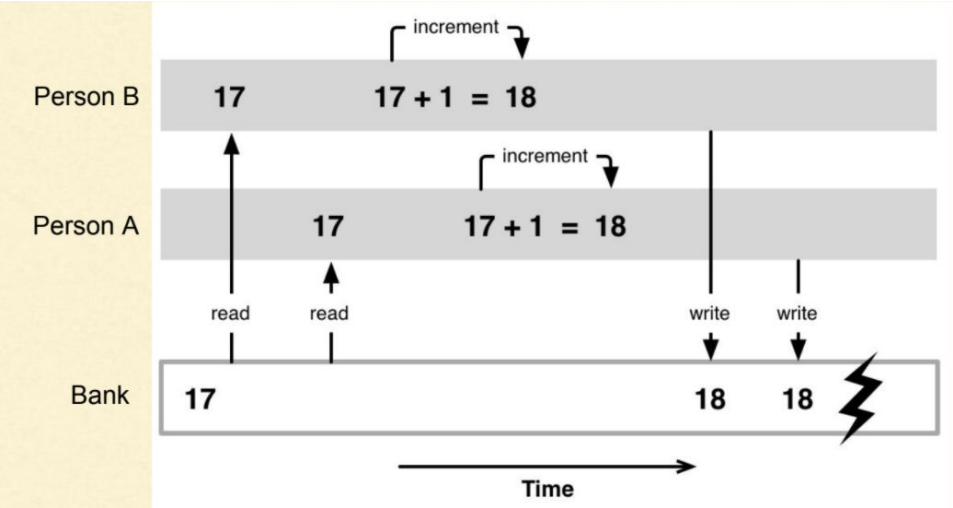
Cuando varios hilos manipulan concurrentemente objetos conjuntos, puede llevar a resultados erróneos o a la paralización de la ejecución.



Problemas de Sincronización

Condición de Carrera

Una condición de carrera ocurre cuando varios hilos acceden y modifican simultáneamente un recurso compartido sin la adecuada sincronización, lo que puede llevar a resultados inesperados o incorrectos.



Problemas de Sincronización

Condición de Carrera

Dos hilos, sumador y restador, se ejecutan al mismo tiempo sobre la misma variable:

- Sumador: cuenta++;
- Restador: cuenta--;

En código máquina eso se representa por:

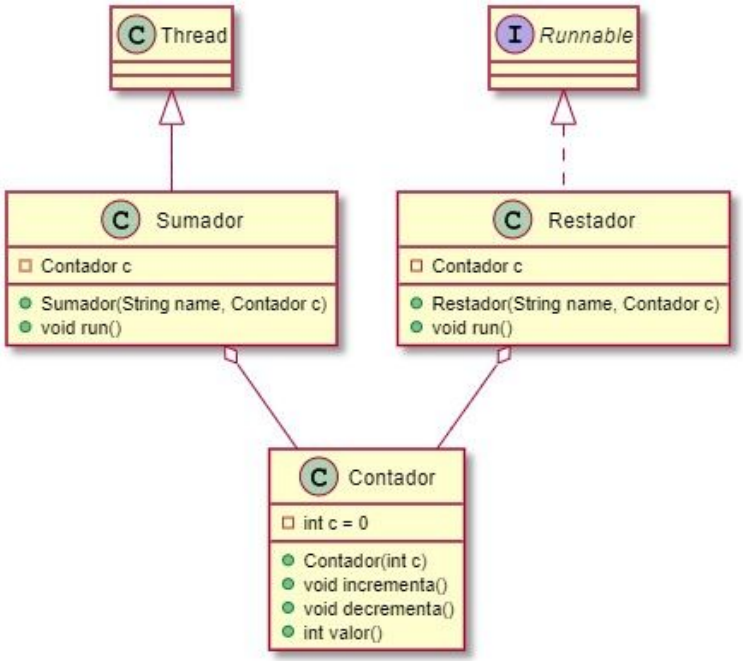
- registroX=cuenta
- registroX = registroX (operación: suma o resta)
- cuenta = registroX

Supongamos que cuenta vale 10, y los dos hilos están ejecutándose. Puede suceder que ambos lleguen a la instrucción que modifica cuenta a la vez y se ejecute lo siguiente:

- T0: sumador → registro1 = cuenta {registro1 = 10}
- T1: sumador → registro1 = registro1 + 1 {registro1 = 11}
- T2: restador → registro2 = cuenta {registro2 = 10}
- T3: restador → registro2 = registro2 - 1 {registro2 = 9}
- T4: sumador → cuenta = registro 1 {cuenta = 11}
- T5: restador → cuenta = registro2 {cuenta = 9}

Problemas de Sincronización

Condición de Carrera



Problemas de Sincronización

Inconsistencia de memoria

Una inconsistencia de memoria se produce cuando diferentes hilos tienen una visión diferente de lo que debería ser el mismo dato.

Ejemplo: Dos hilos, sumador e impresor, realizan el siguiente código partiendo de cuenta=0

- Sumador: cuenta++;
- Impresor: System.out.println(cuenta).

Problemas de Sincronización

Inanición

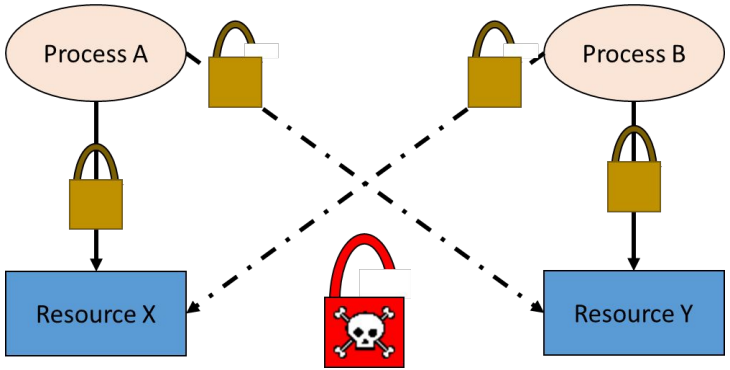
Fenómeno que tiene lugar cuando a un proceso o hilo se le deniega continuamente el acceso a un recurso compartido. Este problema se produce cuando un proceso nunca llega a tomar el control de un recurso debido a que el resto de procesos o hilos siempre toman el control antes que él por diferentes motivos, como por ejemplo, la prioridad.

Problemas de Sincronización

Interbloqueo

Un *interbloqueo* se produce cuando dos o más procesos o hilos están esperando indefinidamente por un evento que solo puede generar un proceso o hilo bloqueado.

- Dicho error no tiene por qué aparecer en todas las ejecuciones realizadas, sino que puede ocurrir únicamente en casos muy concretos que dependen del orden específico en que se ejecuten los hilos.



Problemas de Sincronización

Bloqueo Activo

Un *bloqueo activo* es similar a un interbloqueo, excepto que el estado de los dos procesos envueltos en el bloqueo activo cambia constantemente con respecto al otro.

- En un ejemplo del mundo real, un bloqueo activo ocurre, por ejemplo, cuando dos personas se encuentran en un pasillo avanzando en sentidos opuestos y cada una trata de ser amable moviéndose a un lado para dejar a la otra persona pasar. Si se mueven ambas de lado a lado, encontrándose siempre en el mismo lado, se están moviendo, pero ninguna podrá avanzar.

Monitores y Bloqueos

La sincronización en Java se realiza mediante **monitores**, que son bloqueos asociados a cada objeto.

- Es una propiedad que proporciona la clase *Object*.
- Este mecanismo permite a un único *thread* a la vez ejecutar la sección de código protegida por el monitor → exclusión mutua.
- Los hilos que esperan el bloqueo se gestionan en una cola, pero la selección del siguiente hilo es indeterminada.

Secciones Críticas

En Java la palabra reservada *synchronized* sirve para hacer que un bloque de código o un método sea protegido por el cerrojo del objeto.

- Para ejecutar un bloque o un método sincronizado, los hilos deben conseguir previamente el bloqueo (candado) del objeto, debiendo esperar a que quede libre (el hilo que lo tiene lo libere) si el monitor ya ha sido adquirido por otra hilo.
- Esto ocurre sólo si se está intentando acceder al monitor del mismo objeto que otro hilo ya tenga en propiedad.

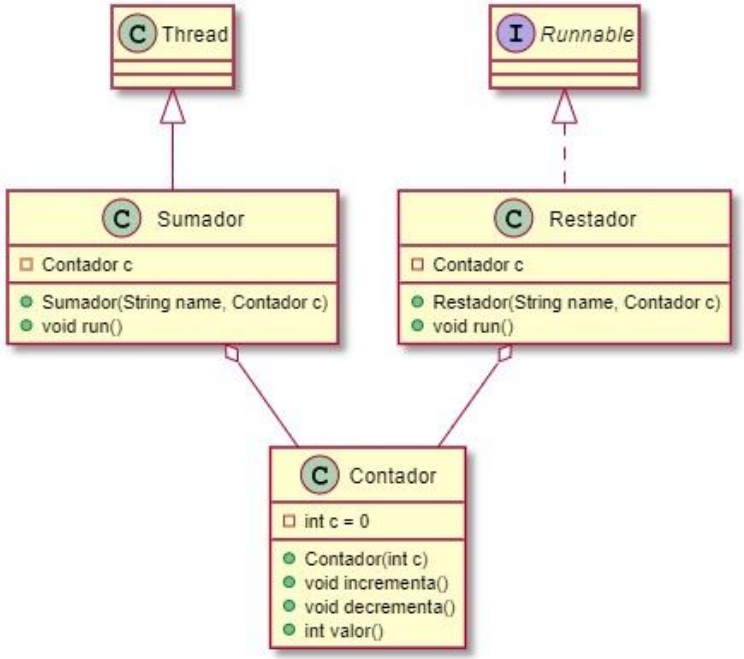
Secciones Críticas

Sincronización de métodos no estáticos

- El bloqueo se aplica sobre el objeto sobre el que se ejecuta el método (*this*). En este caso dos hilos no podrían ejecutar a la vez dos métodos del mismo objeto marcados como *synchronized*.
- Cada objeto que instanciemos de la clase tendrá su propio monitor asociado que no interferirá con los bloqueos que se hayan hecho sobre otros objetos de la misma clase.
- Este comportamiento sólo es válido si todos los métodos sincronizados a los que se quiere acceder pertenecen a la misma instancia.

Secciones Críticas

Sincronización de métodos no estáticos




```
public class Counter {
 private int count = 0;
 public synchronized void add(int value){
   this.count += value;
 }
 public synchronized void sub(int value){
   this.count -= value;
 }
}
```

Secciones Críticas

Sincronización de métodos estáticos

En este caso el bloqueo se realiza sobre la clase a la que pertenece el método. Como sólo hay una instancia de cada objeto clase en la JVM, sólo un hilo a la vez podrá adquirir el monitor y ejecutar el código protegido de una clase estática.

```
class Contador {
     private static int cuenta = 0;
     // Método estático synchronized
     public static synchronized void incrementar() {
          cuenta++;
          System.out.println("Cuenta:" + cuenta + " - " +
          Thread.currentThread().getName());
     }
}
```

Secciones Críticas

Sincronización de porciones de código

La sincronización no tiene porqué realizarse sobre todo un método. Para sincronizar un bloque de código usamos la palabra reservada *synchronized* seguida, entre paréntesis, del objeto del que usaremos el monitor. El código protegido se ubicará entre un par de llaves.

```
public void add(int value){
  synchronized(this){
     this.count += value;
  }
 }
```

Secciones Críticas

Sincronización de porciones de código

```
public class MyClass {
  public synchronized void log1(String msg1, String msg2){
     log.writeln(msg1);
     log.writeln(msg2);
  }

  public void log2(String msg1, String msg2){
     synchronized(this){
     log.writeln(msg1);
     log.writeln(msg2);
     }
  }
 }
```

En términos de sincronización, ambos bloques son totalmente equivalentes.

Por lo tanto, sólo un thread podría ejecutar uno de los dos bloques anteriores al mismo tiempo.

Secciones Críticas

En la programación multihilo, y dentro de la sincronización, un bloque protegido por un monitor nos garantiza que:

- Cuando un hilo entra en un bloque synchronized se actualizará el valor de todas las variables visibles para el hilo.
- Cuando un hilo salga de un bloque synchronized todos los cambios realizados por el hilo se actualizarán en la memoria principal.

Secciones Críticas

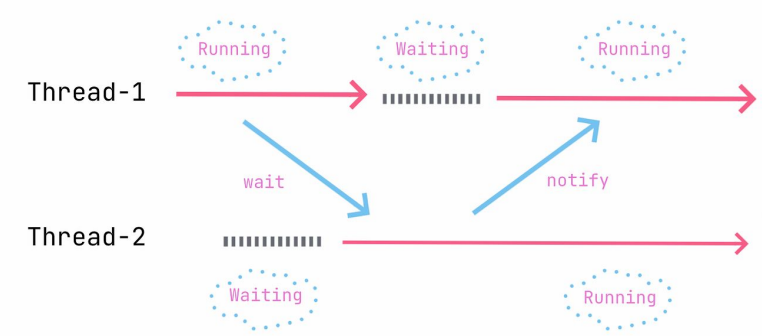
Recomendaciones

- Oracle dice que se puede usar cualquier objeto como monitor de sincronización, sin embargo recomiendan que no se sincronice sobre *String*, o cualquier objeto envoltorio (*wrapper*) de los tipos de datos primitivos (*Integer, Double, Boolean*, ...).
- Un objeto usado como monitor, o como memoria compartida entre hilos, debería ser de tipo *final*, porque si se le asigna un nuevo valor quedan si efecto todos los bloqueos que existan sobre dicho objeto.

Sincronización entre Hilos

A veces no basta con solo bloquear y desbloquear recursos; también necesitamos que los hilos esperen de manera eficiente cuando no pueden continuar su trabajo debido al estado de los recursos.

Para manejar esto, usamos mecanismos de espera no activa. Esto significa que un hilo puede esperar sin ocupar recursos del procesador, simplemente esperando a que otros hilos cambien el estado de los recursos y lo notifiquen.

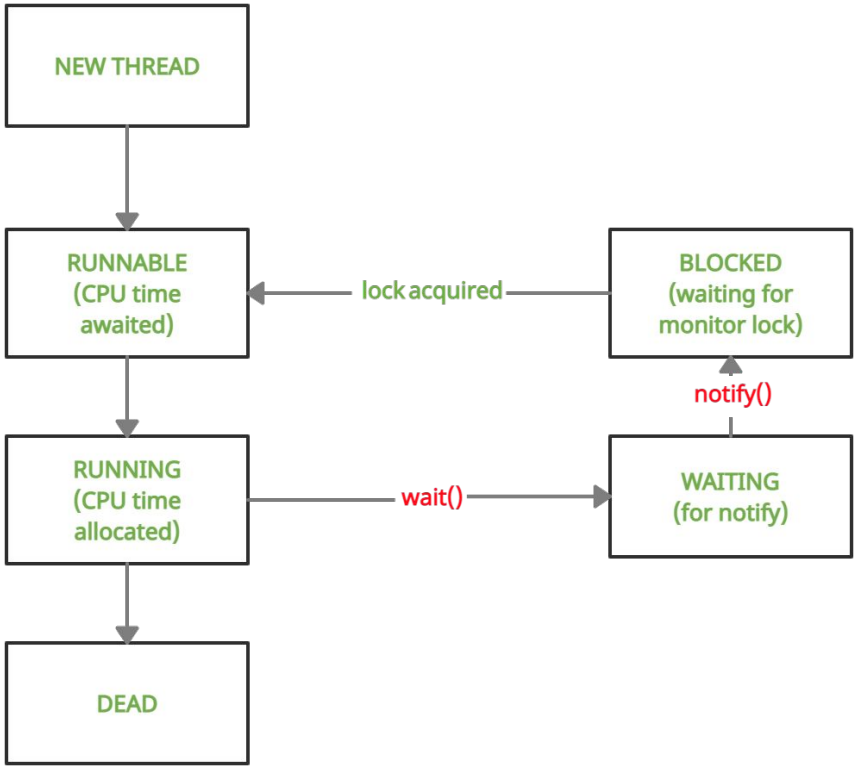


Sincronización entre Hilos

Java proporciona métodos en la clase *Object* para gestionar esta espera:

- *wait():* Un hilo que llama a *wait()* se suspende hasta que otro hilo llame a *notify()* o *notifyAll()* en el mismo objeto. Esto evita que el hilo consuma recursos mientras espera.
- *notify():* Despierta a uno de los hilos que están esperando en el objeto. El hilo despertado puede continuar su ejecución.
- *notifyAll():* Despierta a todos los hilos que están esperando en el objeto. Todos los hilos pueden reanudar su ejecución.

Sincronización entre Hilos



Sincronización entre Hilos

Comparación

- Sin *wait()* (bloqueo activo): El hilo permanece en ejecución y sigue consumiendo ciclos de CPU mientras espera para adquirir el monitor. Esto puede reducir el rendimiento general del sistema, ya que el hilo desperdicia recursos verificando constantemente si puede adquirir el monitor.
- *Con wait() (espera eficiente):* El hilo libera el monitor y entra en estado de **suspensión** hasta que otro hilo lo notifique mediante *notify()* o *notifyAll()*. Durante este tiempo, no consume CPU, lo que hace que la espera sea mucho más eficiente.

Bloqueo Activo

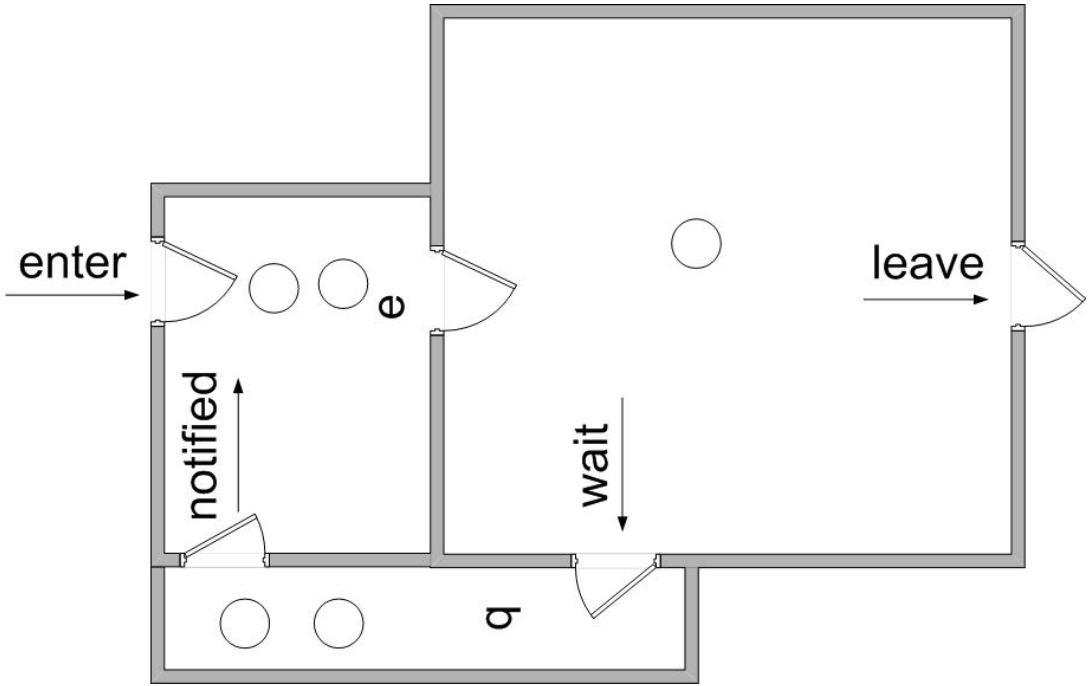
Espera Eficiente

Sincronización entre Hilos

Estos deben llamarse desde un contexto sincronizado, de lo contrario se lanzará una excepción *java.lang.IllegalMonitorStateException*.

- Cuando se llama al método *wait()*, el hilo estará dentro de un bloque sincronizado, por lo tanto tendrá el bloqueo del monitor. En ese momento el hilo libera el bloqueo de ese monitor y se queda en una cola (perteneciente al objeto) de hilos en espera de ser notificados, diferente a la de los hilos que están esperando por el bloqueo.
- Cuando se desbloquea un hilo porque otro ha llamado a *notify()/notifyAll()*, el hilo vuelve al punto donde hizo el *wait()*, por lo tanto sigue dentro de un bloque sincronizado. Para poder continuar con la ejecución tendrá que pasar a la cola de hilos esperando por el bloqueo y esperar a ser seleccionado para seguir ejecutándose.

Sincronización entre Hilos

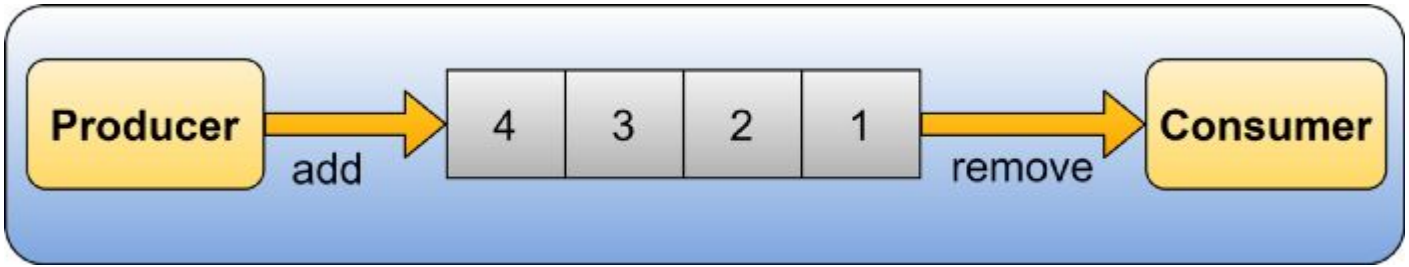


Modelo Productor-Consumidor

El modelo *Productor-Consumidor* es un patrón común en programación multihilo

- Productor genera datos y los coloca en una cola
- Consumidor toma esos datos de la cola para procesarlos.

La idea es coordinar la interacción entre los dos, de modo que el Productor no llene la cola más allá de su capacidad, y el Consumidor no intente consumir cuando la cola está vacía.



Modelo Productor-Consumidor

- Se define un buffer de enteros mediante una cola circular.

```
class Cola {
   private int [] _datos;
   private int _sigEnt, _sigSal, _ocupados, _tamano;

   public Cola ( int tam ) {
       _datos = new int [ tam ];
       _tamano = tam;
       _ocupados = 0;
       _sigEnt = 1;
       _sigSal = 1;
   }

   public synchronized void almacenar ( int x ) {...}

   public synchronized int extraer () {...}
}
```

Declaraciones típicas para esta Estructura de Datos

Antes de realizar la operación se va a asegurar de que no hay otro hilo usando el objeto Cola.

Modelo Productor-Consumidor

```
public synchronized void almacenar ( int x ) {
    try {
        while ( _ocupados == _tamano ) wait ();
        _datos [ _sigEnt ] = x;
        _sigEnt = ( _sigEnt + 1 ) % _tamano;
        _ocupados++;
        notify();
    }
    catch ( InterruptedException e ) {}
}
```

No hay espacio → esperar

Insertar en el buffer

¡He terminado! Despertar a un hilo que esté esperando

`wait` puede lanzar Excepción

```
public synchronized int extraer () {
    int x = 0;
    try {
        while ( _ocupados == 0 ) wait();
        x = _datos [ _sigSal ];
        _sigSal = ( _sigSal + 1 ) % _tamano;
        _ocupados--;
        notify();
    }
    catch ( InterruptedException e ) {}
    return x;
}
```

No hay datos → esperar

Extraer del buffer

¡He terminado! Despertar a un hilo que esté esperando

Modelo Productor-Consumidor

```
//CLASE PRODUCTOR, ACCEDE PARA ALMACENAR
//Genera datos (int) y los almacena en
//el buffer indefinidamente.
class Productor extends Thread {

    private Cola _buffer;

    public Productor ( Cola c ) {
        _buffer = c;
    }

    public void run () {
        int valor = 0;
        while ( true ) {
            _buffer.almacenar ( valor );
            valor++;
        }
    }
}
```

El buffer donde va a almacenar, se asigna en el constructor.

Como el buffer está monitorizado, la tarea se despreocupa del sincronismo con otras tareas.

Modelo Productor-Consumidor

```
//CLASE CONSUMIDOR, ACCEDE PARA EXTRAER
//Extrae datos (int) del buffer indefinidamente.
class Consumidor extends Thread {

    private Cola _buffer;

    public Consumidor ( Cola c ) {
        _buffer = c;
    }

    public void run () {
        int dato;
        while ( true ) {
            dato = _buffer.extraer ();
            System.out.println( dato );
        }
    }
}
```

El buffer donde va a almacenar, se asigna en el constructor.

Modelo Productor-Consumidor

```
public static void main (String [] args) {
//...
    el_buffer = new Cola (5);
///...
}

class Consumidor extends Thread {
//...
    public void run () {
        int dato;
        while ( dato < 10 ) {
            dato = _buffer.extraer ();
            System.out.println( dato );
        }
    }
}
```

```
Productor 0
Productor 1
Productor 2
Productor 3
Productor 4
Productor 5
Consum0 : 0
Consum0 : 1
Consum0 : 2
Productor 6
Consum1 : 3
Consum2 : 4
Consum0 : 5
Productor 7
Consum1 : 6
Consum2 : 7
Productor 8
Consum1 : 8
Productor 9
Consum1 : 9
Consum0 : 10
```

```
Consum0 : 10
FIN: Consum0
Productor 10
Productor 11
Productor 12
Productor 13
Consum1 : 11
FIN: Consum1
Consum2 : 12
FIN: Consum2
Productor 14
Productor 15
Productor 16
Productor 17

Kill
```