

Capítulo 4

- **Static**
- **Singleton**
- **Wrapper Classes**
- **Threads**

Static

- Hay ocasiones en las cuales se desea que un determinado **atributo** o **método** de una clase sea compartido por **todas** las instancias de esa clase.

Ejemplo:

Para una cierta clase podríamos querer tener un **contador** de la cantidad de instancias que han sido creadas para la clase.

- Lo anterior puede lograrse etiquetando al miembro (atributo o método) en cuestión mediante la palabra reservada **static**.
- Un miembro etiquetado como **static** no está asociado a ninguna instancia de la clase en particular, sino que es accesible desde **todas** las instancias de la clase. Comúnmente se dice que los miembros estáticos pertenecen **a la clase** en lugar de a los objetos.
- Solamente podemos declarar como **static** a los atributos y a los métodos de una clase. No podemos declarar parámetros ni variables locales a métodos como **static**.

Static

Atributos Static

Ejemplo: Presentamos una clase con un **contador** de la cantidad de instancias que han sido creadas para la clase.

```
public class Producto
{
    private long codigo;
    private double precio;
    private static int contador;

    public Producto (long codigo, double precio)
    {
        this.codigo = codigo;
        this.precio = precio;
        contador++;
    }

    ...
}
```

Static

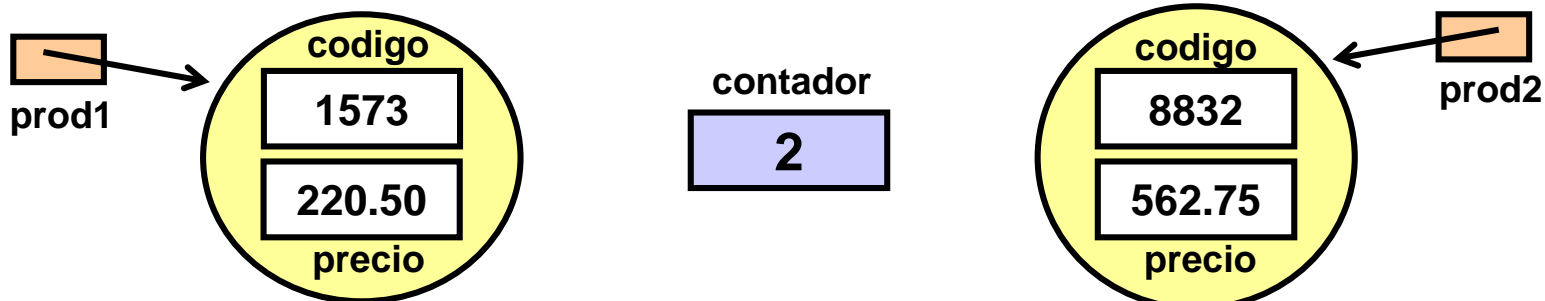
Atributos Static (continuación)

- Creamos ahora dos instancias de la clase **Producto** presentada en el ejemplo anterior:

```
Producto prod1 = new Producto(1573, 220.50);  
// el contador de la clase queda con el valor 1
```

```
Producto prod2 = new Producto(8832, 562.75);  
// el contador de la clase queda con el valor 2
```

- Cada instancia de la clase **Producto** posee sus propios atributos **código** y **precio**, pero el atributo **contador** **no** pertenece a ninguno de ellos, sino que pertenece **a la clase**.



Static

Métodos Static

Ejemplo: Le agregamos un método estático a la clase `Producto`

```
public class Producto
{
    private long codigo;
    private double precio;
    private static int contador;

    ...

    public long getCodigo()
    { return codigo; }

    public double getPrecio()
    { return precio; }

    public static int cantInstancias()
    { return contador; }
}
```



¿ Puedo acceder a un **atributo static** desde un **método común** ?

¿ Puedo acceder a un **atributo común** desde un **método static** ?

Static

Métodos Static (continuación)

- A diferencia de los métodos comunes, que se invocan a través del nombre de un **objeto**, los métodos estáticos se invocan a través del nombre de la **clase**. Veamos un ejemplo:

// invocamos métodos comunes

```
long cod1 = prod1.getCodigo(); // retorna 1573
long cod2 = prod2.getCodigo(); // retorna 8832
double pre1 = prod1.getPrecio(); // retorna 220.50
double pre2 = prod2.getPrecio(); // retorna 562.75
```

// invocamos el método estático

```
int count = Producto.cantInstancias(); // retorna 2
```

- No es la primera vez que vemos invocaciones de este tipo. Por ejemplo, `Math.random()` o `System.out.println()` son invocaciones a métodos estáticos.

Singleton

- Hay ciertas ocasiones en Orientación a Objetos en las cuales se desea tener una **única** instancia de una determinada clase y que la misma sea accedida fácilmente desde cualquier punto dentro de la aplicación. Por ejemplo:
 - ❖ Queremos tener una única instancia de la clase que se conecta con una base de datos.
 - ❖ Queremos tener una única instancia de la clase que se comunica con el programa servidor de nuestra aplicación.
- Lo anterior es fácilmente realizable en Java mediante la utilización de un mecanismo popularmente conocido como **Singleton**. Dicho mecanismo no es más que una forma **ingeniosa** de programar nuestra clase para garantizar la existencia de una única instancia de ella.

Nota: Formalmente hablando, **Singleton** es en realidad un ***Patrón de Diseño***. Los patrones de diseño son descripciones de soluciones pre-establecidas a problemas que se repiten una y otra vez en Orientación a Objetos. A lo largo de la evolución de la P.O.O muchos desarrolladores han ido inventando y documentando Patrones en libros o tutoriales para que otros desarrolladores puedan aprenderlos y utilizarlos en sus propios diseños.

Singleton

Ejemplo: Presentamos una clase implementada de forma tal que aplica el mecanismo **Singleton**.

```
public class MiClase
{
    private static MiClase instancia;
    // otros atributos, pueden ser comunes o estáticos

    private MiClase()
    { ... }

    public static MiClase getInstancia()
    {
        if (instancia == null)
        {
            instancia = new MiClase();
        }

        return instancia;
    }
    ...
    // otros métodos, pueden ser comunes o estáticos
}
```


Singleton

- Veamos una descripción del ejemplo anterior:
- **Método constructor**: Dado que **no** queremos que se puedan crear múltiples instancias de la clase, etiquetamos el método constructor como **privado**. De este modo **no** podrá ser invocado desde fuera de la clase y por lo tanto **no** se podrán crear múltiples instancias de la misma.
- **Atributo instancia**: Necesitamos tener alguna **referencia** a la única instancia que existirá de la clase. La declaramos **privada** a efectos de evitar un manejo descuidado de la misma desde fuera de la clase.
- **Método `getInstancia()`**: Debemos tener algún método que permita retornar la única instancia de la clase a quien la solicite. Lo declaramos **static** para que sea fácilmente invocable (a través del nombre de la clase) desde cualquier lugar de nuestra aplicación. La primera vez que sea invocado, creará la instancia y la retornará. En sucesivas invocaciones, solamente la retornará, garantizando así su unicidad.

Wrapper Classes

- Por defecto, Java **no** considera como Objetos a los valores de tipos de datos primitivos. Sin embargo, hay contextos en los cuales se necesita trabajar con dichos valores como si fueran objetos.
- Para hacer esto, Java provee unas clases llamadas **Wrapper Classes** (definidas en `java.lang`) las cuales toman un valor de un tipo primitivo, y lo “envuelven” en un Objeto. Hay una **Wrapper Class** para cada tipo primitivo:

<code>int</code>	→	<code>Integer</code>
<code>double</code>	→	<code>Double</code>
<code>char</code>	→	<code>Character</code>
<code>boolean</code>	→	<code>Boolean</code>
<code>long</code>	→	<code>Long</code>

Ejemplo:

```
Integer objNum1 = new Integer(50);  
// Transforma al valor 50 en un objeto  
Integer objNum2 = 50;  
// Hace lo mismo, pero de forma implícita
```

Wrapper Classes

- Las Wrapper Classes poseen muchos métodos de utilidad a la hora de hacer conversiones entre valores de tipos primitivos y objetos.
- Por ejemplo, el siguiente método de la clase `Integer` sirve para transformar un `String` en un valor entero.

```
public static int parseInt (String s)  
// Transforma al string recibido en un valor entero. Si la cadena  
// de caracteres del string no representa un valor entero, lanza  
// una excepción
```

Ejemplo: `int numero = Integer.parseInt("1234");`
 `// convierte el string "1234" al entero 1234`

- Todas las Wrapper Classes poseen métodos equivalentes que permiten transformar a `String` un valor del tipo correspondiente.

Threads

- Ya sabemos que Java es un lenguaje multitarea. Esto significa que podemos programar rutinas que se ejecuten en forma **paralela**. Cada una de esas rutinas **concurrentes** se conoce como **Thread** (hilo) y está compuesta por:
 - Una **CPU** virtual (ejecuta las instrucciones propias de la rutina)
 - Una sección de **código** (instrucciones a ser ejecutadas por la CPU virtual en forma concurrente con otras rutinas)
 - Una sección de **datos** (variables y objetos utilizados por la rutina)
- Cuando comienzan a ejecutarse dos o más threads, la CPU virtual de cada uno ejecuta las instrucciones de la sección de código correspondiente.
- A su vez, la **JVM** hace las veces de **despachador** (scheduler) conmutando de un thread a otro durante la ejecución del mismo modo que lo haría el scheduler de cualquier Sistema Operativo.

Threads

- Veremos que dos threads pueden compartir la **misma** sección de **código**, siendo ejecutada por la CPU virtual de cada uno sin que esto afecte al otro.
- Veremos también que dos threads pueden compartir la **misma** sección de **datos**, en cuyo caso habrá que tener ciertos cuidados para preservar la consistencia de esos datos.

Creación de Threads

- Para programar un thread en Java tenemos **dos** alternativas:
 - **Implementar** una interface predefinida llamada **Runnable**
 - **Extender** una clase predefinida llamada **Thread**
- Ambas alternativas son **equivalentes** en cuanto a su desempeño. Sin embargo, la primera alternativa es **preferible** porque permite un mejor diseño Orientado a Objetos.

Referencia: API – `java.lang.Runnable` y `java.lang.Thread`

Threads

Creación de Threads (continuación)

- Si decidimos crear un Thread **implementando** `Runnable` tenemos que:
 - Escribir una **clase** que implemente dicha interface.
 - En ella, implementar el único método de la interface, llamado `run()`.
- Si decidimos crear un Thread **extendiendo** `Thread` tenemos que:
 - Escribir una **clase** derivada de `Thread`.
 - En ella, sobre-escribir el método `run()` de la clase base.
- Ya sea de una u otra forma, lo que incluiremos dentro del método `run()` es la sección de código del Thread. Es decir, las instrucciones a ser ejecutadas en forma concurrente con otros Threads. En este curso vamos a centrarnos **solamente** en la primera alternativa, dado que la misma es preferible.

Observación: La clase `Thread` de por sí implementa `Runnable`. Lo que hace es simplemente dar un cuerpo vacío `{ }` al método `run()`.

Threads

Creación de Threads (continuación)

Ejemplo:

```
// HiloAlfa.java
public class HiloAlfa implements Runnable
{
    public void run()
    {
        for (int i = 0; i < 200; i++)
            System.out.println ("Soy el hilo alfa: " + i);
    }
}

// HiloBeta.java
public class HiloBeta implements Runnable
{
    public void run()
    {
        for (int i = 0; i < 200; i++)
            System.out.println ("Soy el hilo beta: " + i);
    }
}
```

Threads

Creación de Threads (continuación)

- Ahora tenemos que poner a correr ambos hilos concurrentemente. Para ello, debemos crear una instancia de la clase **Thread** por cada uno e invocarles un método llamado **start()**, el cual deja a los hilos listos para que la JVM empiece a conmutar entre ellos cuando lo disponga. (**NO** invocamos directamente al método **run()**)

```
// PruebaHilos.java
public class PruebaHilos
{
    public static void main (String args [])
    {
        Thread t0 = new Thread (new HiloAlfa());
        Thread t1 = new Thread (new HiloBeta());
        t0.start();
        t1.start();
    }
}
```


Threads

Creación de Threads (continuación)

- La ejecución concurrente del método `run()` en cada uno de los Threads ocasionará que los mensajes desplegados en pantalla aparezcan **intercalados**. No obstante, es imposible **predecir** cuál será el orden de intercalación, dado que la JVM puede variar sus estrategias de conmutación de una ejecución a otra.

```
Soy el hilo alfa: 0
Soy el hilo alfa: 1
Soy el hilo alfa: 2
Soy el hilo beta: 0
Soy el hilo alfa: 3
Soy el hilo beta: 1
Soy el hilo beta: 2
Soy el hilo alfa: 4
Soy el hilo alfa: 5
...
```

Threads

Threads que comparten código

- En los ejemplos anteriores, cada uno de los threads poseía **su propia sección de código**. Sin embargo, en ocasiones podemos necesitar que dos (o más) threads realicen una **misma** tarea. Para ello necesitamos que ambos threads **compartan** la misma sección de código. Esto se logra creando dos instancias de la clase Thread a partir del mismo método `run()`.

Ejemplo: Creamos una sola clase que implemente el método `run()`.

```
// MismoCodigo.java
public class MismoCodigo implements Runnable
{
    public void run()
    {
        for (int i=0; i<200; i++)
        {
            String name = Thread.currentThread().getName();
            System.out.println ("Soy el hilo: " + name);
        }
    }
}
```

Threads

Threads que comparten código (continuación)

Ejemplo: Creamos dos Threads que ejecutan el mismo método `run()`.

```
// PruebaMismoCodigo.java
public class PruebaMismoCodigo
{
    public static void main (String args [])
    {
        MismoCodigo cod = new MismoCodigo();
        Thread t0 = new Thread (cod);
        Thread t1 = new Thread (cod);
        t0.start();
        t1.start();
    }
}
```

Observación: Si bien ambos threads ejecutan el mismo código, la ejecución de uno no interfiere con la del otro porque cada thread ejecuta el método `run()` usando su propia CPU virtual.

Threads

Threads que comparten datos

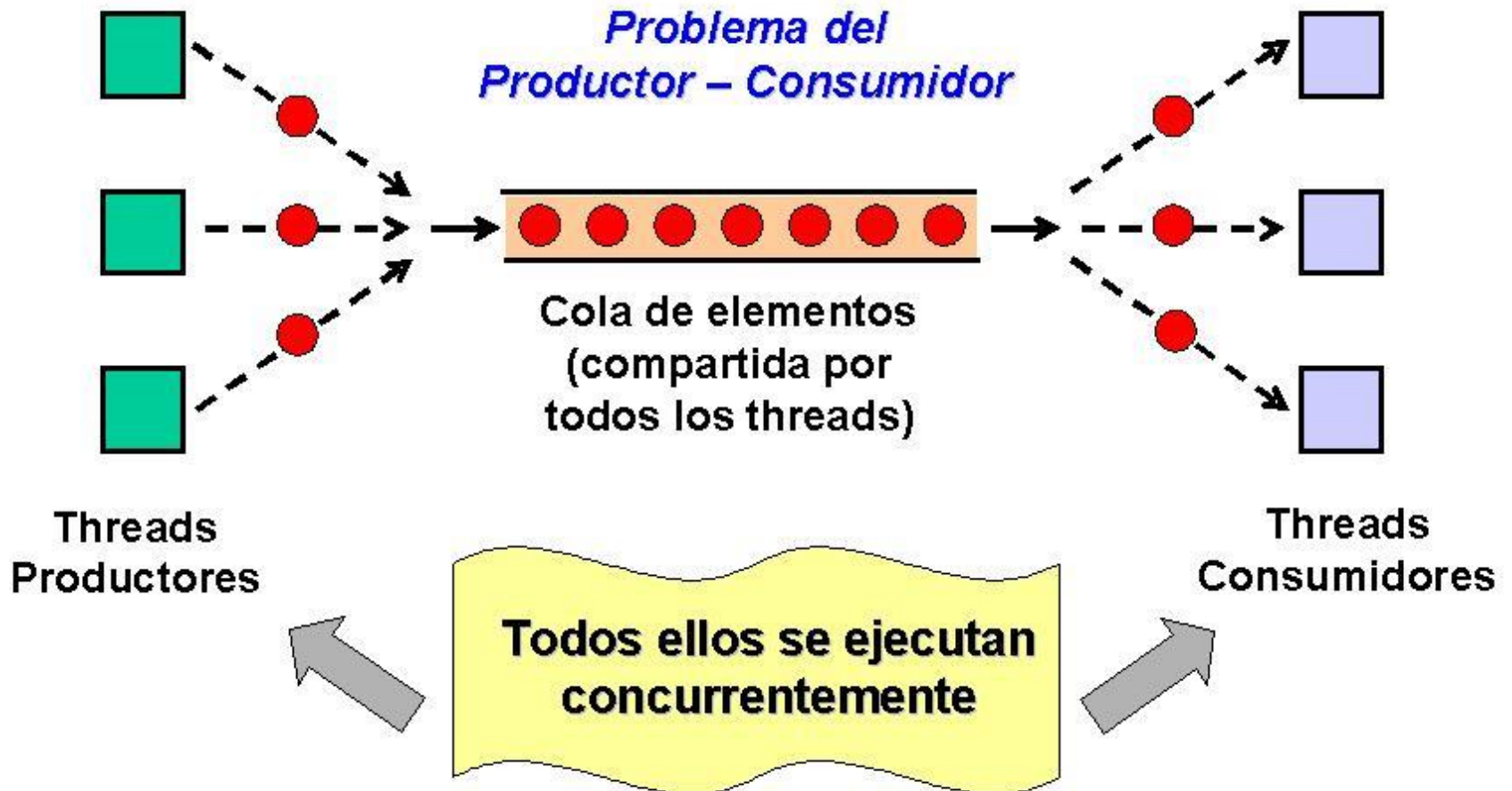
- Hay situaciones en las cuales dos (o más threads) deben compartir los mismos **datos** durante su ejecución en forma concurrente.

Ejemplo: Consideremos el clásico problema de concurrencia denominado ***Problema del Productor – Consumidor.***

- Se tiene una **Cola** (Queue) que almacena elementos de algún tipo.
 - Se tienen uno o más hilos **productores**, cuya función es producir elementos e insertarlos en la cola.
 - Se tienen uno o más hilos **consumidores**, cuya función es quitar elementos de la cola para consumirlos (darles algún tratamiento).
- En este ejemplo, los procesos productores y consumidores se ejecutan en forma paralela. Todos ellos intentan acceder **concurrentemente** a la Cola de elementos, la cual es compartida por todos ellos. En definitiva, la Cola constituye el conjunto de datos compartidos por todos los hilos.

Threads

Threads que comparten datos (continuación)



Threads

Threads que comparten datos (continuación)

- En Java, los datos compartidos deben **encapsularse** en un objeto que los englobe. Los diferentes threads que deseen acceder concurrentemente a dichos datos invocarán a los métodos del objeto que los encapsula.
- No obstante, debemos garantizar que el acceso concurrente a los datos compartidos se haga de forma **controlada**, a fin de preservar la integridad y consistencia de dichos datos.
- Para ello, es necesario que cada método se ejecute en forma **atómica**, es decir, como si fuera **una** sola instrucción. En Java, ello se logra fácilmente etiquetando cada método del objeto compartido como **synchronized**.
- Cuando un thread invoque a un método sincronizado del objeto compartido, se **adueñará** del mismo mientras dure la invocación. En el interín, si la JVM intenta despachar **otro** thread que pretenda acceder también al objeto compartido, el mismo será **bloqueado** hasta que el thread original finalice la ejecución del método sincronizado.

Threads

Threads que comparten datos (continuación)

Ejemplo:

```
public class DatosCompartidos
{
    // atributos de la clase
    ...

    public synchronized void método1 (...)
    {
        // El método1 se ejecutará atómicamente
    }

    public synchronized int método2 (...)
    {
        // El método2 se ejecutará atómicamente
    }

    ...
}
```

Threads

Threads que comparten datos (continuación)

- Hay situaciones en las cuales un método del objeto compartido no puede terminar de ejecutarse porque no cuenta con todos los recursos necesarios para hacerlo.

Ejemplo: Considere nuevamente el problema del Productor – Consumidor

- ¿Qué ocurre si un productor intenta depositar un elemento en la cola cuando se encuentra **llena**?
 - ¿Qué ocurre si un consumidor intenta quitar un elemento en la cola cuando se encuentra **vacía**?
- En estos casos, el thread que está ejecutando el método deberá **esperar** hasta que otro thread acceda al objeto compartido y le provea los recursos necesarios para que el thread original pueda terminar de ejecutar el método.

Threads

Threads que comparten datos (continuación)

- El problema en estos casos es que el thread original se encuentra ejecutando un método **sincronizado** del objeto compartido. Esto significa que se ha **adueñado** del mismo y por lo tanto ningún otro thread podrá accederlo con el fin de proveer los recursos necesarios al thread original.
- Para solucionar este inconveniente, el thread original puede pedirle a la JVM ser **retirado** de la ejecución hasta que algún otro thread provea al objeto compartido de los recursos que él necesita.
- Al hacerlo, dejará al objeto compartido **disponible** para que algún otro thread pueda accederlo y pasará a un estado de **bloqueo**.
- Una vez que otro thread haya provisto tales recursos, debe **notificar** a la JVM que lo ha hecho, a fin de que disponga el desbloqueo del thread original. Cuando a dicho thread le toque nuevamente ser puesto en ejecución por la JVM, retomará la ejecución del método sincronizado en la misma línea en la cual lo abandonó.

Threads

Threads que comparten datos (continuación)

- Los siguientes métodos se encuentran definidos en la clase `Object` (y por lo tanto son heredados por la clase que encapsula los datos compartidos). Sirven para bloquear y desbloquear a los distintos threads que los invocan sobre el objeto compartido.

```
public final void wait() throws InterruptedException
// Bloquea al thread que lo invoca, liberando así el objeto sobre el cual
// es invocado. El thread podrá ser desbloqueado sólo cuando otro
// thread invoque al método notify() sobre el mismo objeto.
```

```
public final void notify()
// Notifica a la JVM que debe despertar a algún thread que haya
// solicitado bloquearse sobre el objeto para el cual es invocado. La
// JVM des-bloqueará un thread elegido al azar. Si no había ninguno
// bloqueado, la operación no tiene efecto.
```

Threads

Threads que comparten datos (continuación)

➤ Observaciones:

- La etiqueta **final** aplicada a un método impide que el mismo pueda ser sobre-escrito en una clase derivada. Esto significa que los métodos **wait()** y **notify()** **no** pueden ser sobre-escritos.
- Los métodos **wait()** y **notify()** deben ser invocados **sólo** desde dentro de bloques **sincronizados**. En caso contrario, se lanza una **Runtime Exception**.
- Hay distintas variantes para los métodos **wait()** y **notify()** en la clase **Object**, pero todas ellas tienen un comportamiento similar.
- La excepción declarada en el cabecal del método **wait()** no es una **Runtime Exception**. Esto significa que debe ser atrapada con un bloque **try – catch**. La misma es lanzada si el thread bloqueado es interrumpido por otro thread en forma imprevista durante el bloqueo, en vez de ser despertado adecuadamente mediante **notify()**.

Threads

Un Ejemplo Completo

- Presentamos una solución en Java al problema del Productor – Consumidor introducido anteriormente.
- Por simplicidad del ejemplo, suponemos que la Cola (el objeto compartido) es **no acotada**. Dado que la misma es una **secuencia** de elementos que cambia constantemente de tamaño, utilizaremos una **LinkedList** para almacenarlos.
- En el ejemplo utilizaremos valores enteros (Objetos de la clase **Integer**) para almacenar en la cola.
- Dado que existe **una** sola instancia de la Cola, aplicaremos **Singleton** en su implementación.

Threads

La clase Cola

```
// Cola.java
import java.util.LinkedList;
public class Cola
{
    private LinkedList<Integer> lista;
    private static Cola instancia; // Singleton !!

    private Cola ()
    { lista = new LinkedList<Integer> (); }

    public synchronized static Cola getInstancia ()
    { if (instancia == null)
        instancia = new Cola(); // Singleton !!
        return instancia;
    }

    ...
}
```

Threads

La clase Cola (continuación)

```
...  
public synchronized Integer quitar()  
{  
    while (lista.isEmpty())  
    {  
        // espero hasta que la cola no esté vacía  
        try  
        {    this.wait(); }  
        catch (InterruptedException e)  
        {    // no hago nada  
        }  
    }  
  
    // quito un entero de la cola y lo devuelvo  
    return lista.removeFirst();  
}  
...
```

Threads

La clase Cola (continuación)

```
...  
public synchronized void insertar (Integer num)  
{  
    // inserto un valor en la cola y notifico a  
    // algún consumidor que esté bloqueado esperando  
    // que la cola no esté vacía  
    lista.add(num) ;  
    this.notify() ;  
}  
} // fin de la clase Cola
```

Threads

La clase Productor

```
// Productor.java
public class Productor implements Runnable
{
    public void run()
    {
        Cola cola = Cola.getInstance();

        for (int i = 0; i < 200; i++)
        {
            // inserto 200 elementos al azar en la cola
            Integer num = (int) (Math.random() * 100);
            cola.insertar (num);
            System.out.println ("Productor: " + num);
        }
    }
}
```


Threads

La clase Consumidor

```
// Consumidor.java
public class Consumidor implements Runnable
{
    public void run()
    {
        Cola cola = Cola.getInstance();

        for (int i = 0; i < 200; i++)
        {
            // quito 200 elementos de la cola
            Integer num = cola.quitar();
            System.out.println ("Consumidor: " + num);
        }
    }
}
```

Threads

La clase PruebaProdCons

```
// PruebaProdCons.java
public class PruebaProdCons
{
    public static void main (String args [])
    {
        Productor p = new Productor();
        Consumidor c = new Consumidor();

        Thread prod1 = new Thread (p);    // creo dos
        Thread cons1 = new Thread (c);    // productores
        Thread prod2 = new Thread (c);    // y dos
        Thread cons2 = new Thread (c);    // consumidores

        prod1.start();
        cons1.start();                    // los pongo a correr
        prod2.start();                    // concurrentemente
        cons2.start();

    }
}
```