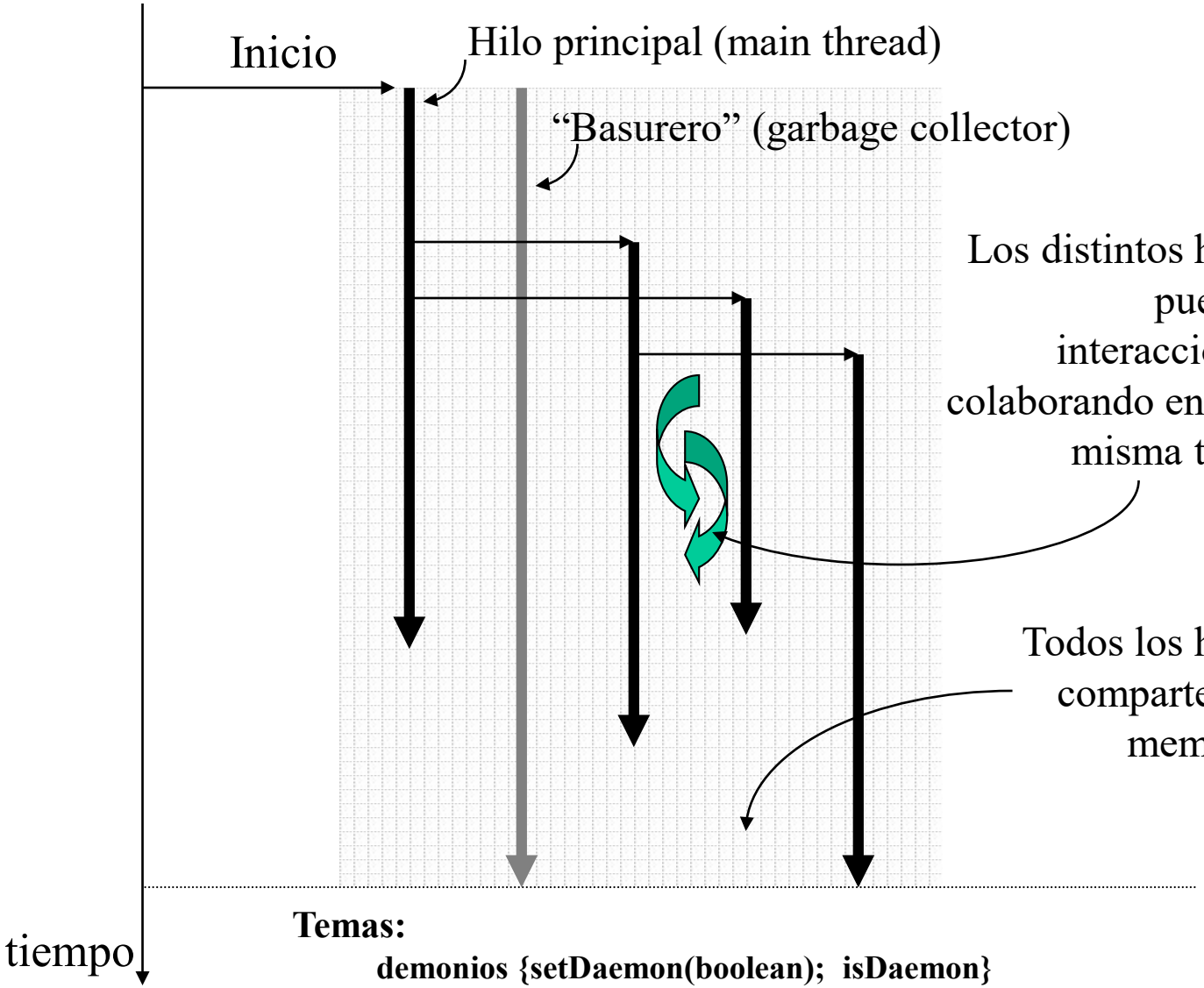
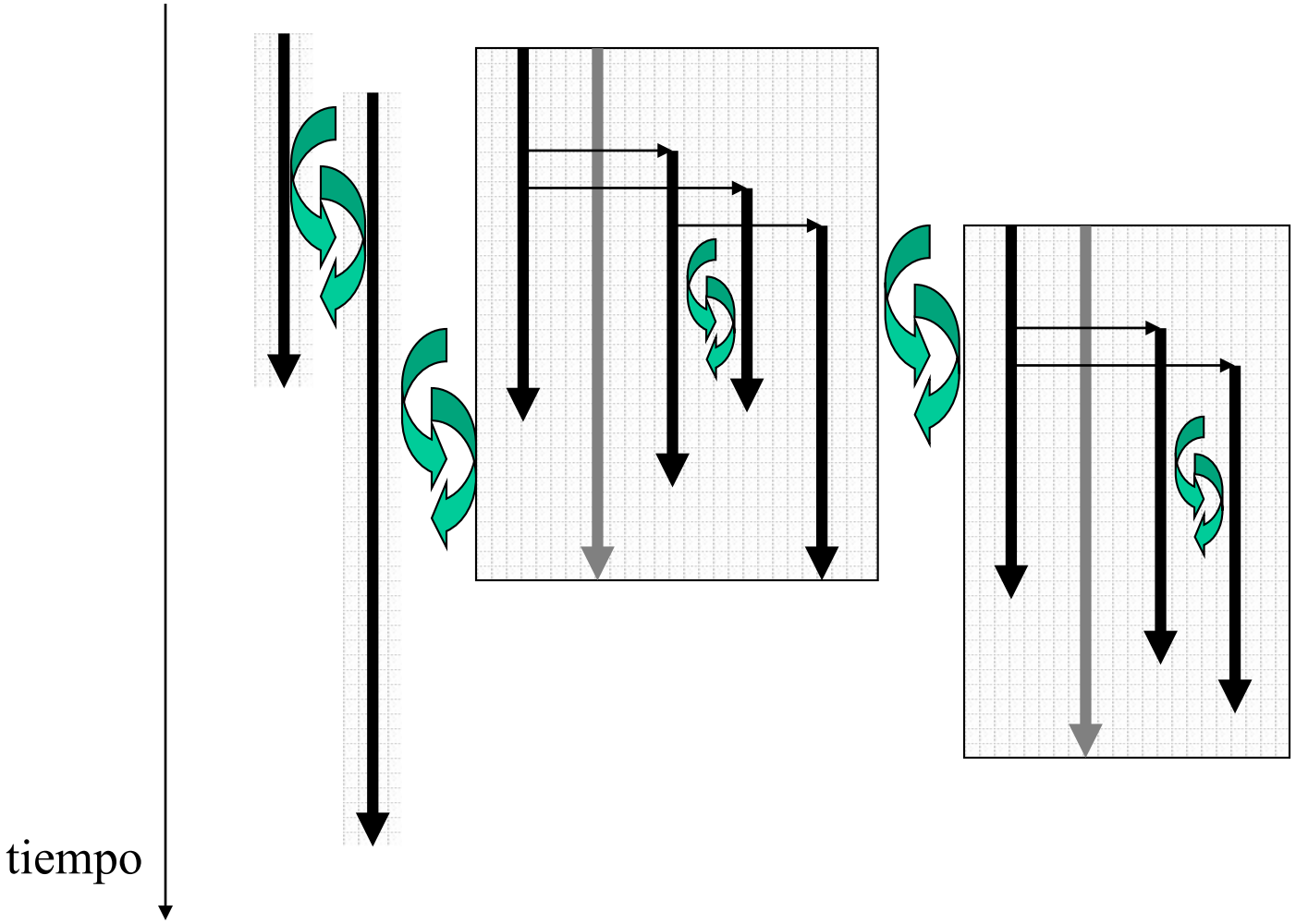
Threads (Hilos)

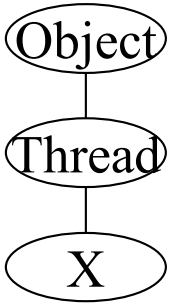


Temas:

demonios {setDaemon(boolean); isDaemon}
prioridades y "scheduling" {setPriority(int); getPriority()}
secciones críticas {synchronized}
sincronización

Procesos en un S.O.





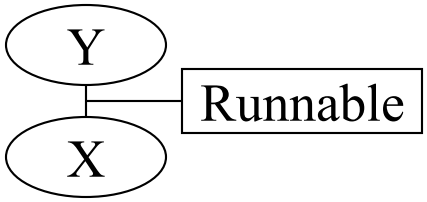
Objeto de subclase de Thread

```
class X  extends Thread {
.....

public void run()
 { // código origen del hilo
  }
}
```

```
X a = new X(); a.start();
```

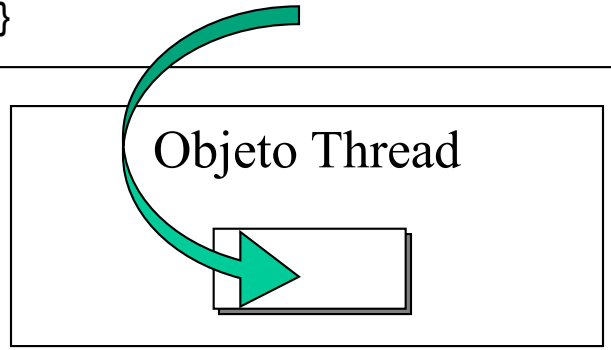
*El start() puede situarse en el constructor



Objeto de clase Runnable

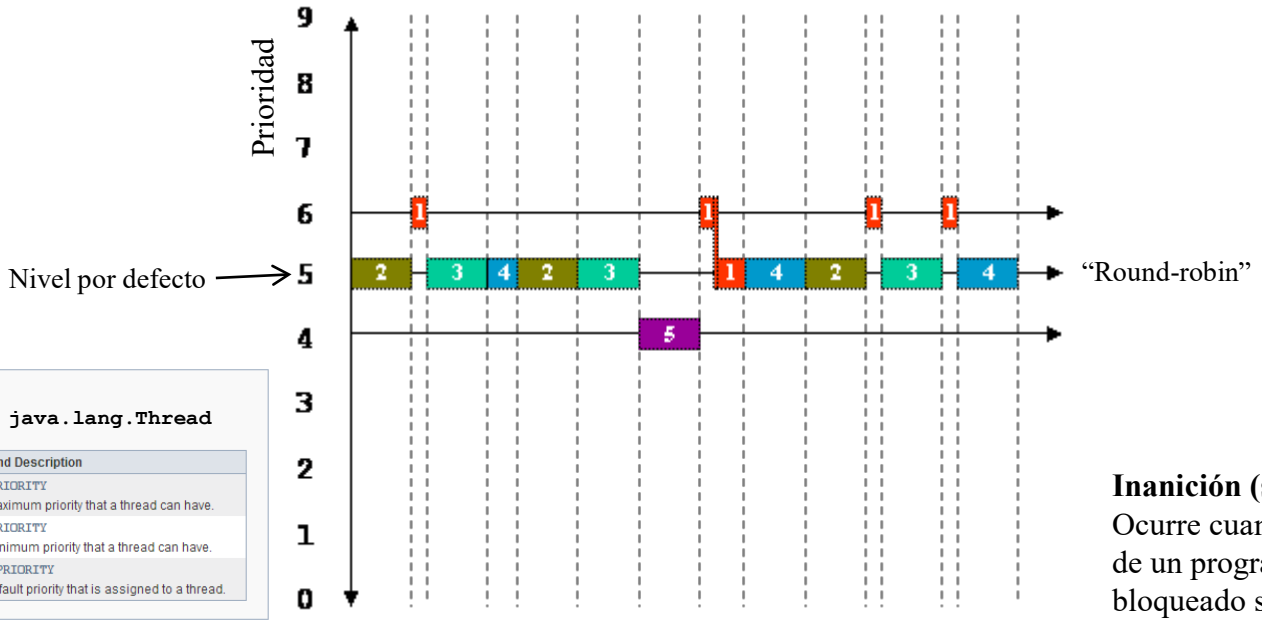
```
class X  extends Y implements Runnable {
.....

public void run()
 { // código origen del hilo
  }
}
```




```
X a = new X(); Thread t=new Thread(a); t.start();
```

Hilos – "Scheduling"
asignación de tiempos y Prioridades



Field Summary

java.lang.Thread

Fields

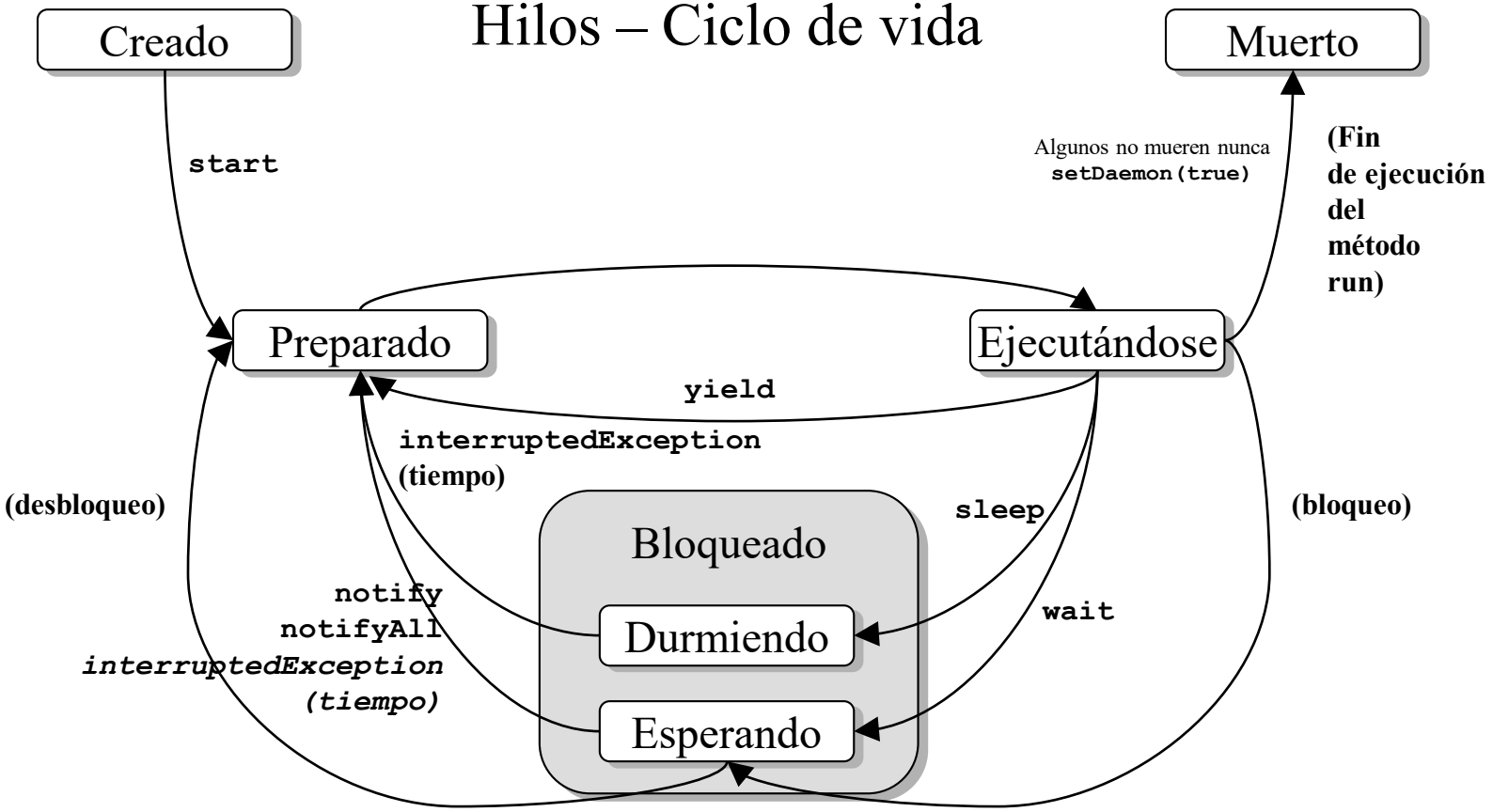
Modifier and Type	Field and Description
static int	MAX_PRIORITY The maximum priority that a thread can have.
static int	MIN_PRIORITY The minimum priority that a thread can have.
static int	NORM_PRIORITY The default priority that is assigned to a thread.

java.lang.Thread

- `public final void setPriority(int newPriority)`
- `public final int getPriority()`

Inanición (starvation)
Ocurre cuando uno o más hilos de un programa ven siempre bloqueado su acceso a un recurso y por tanto no pueden progresar

Hilos – Ciclo de vida



java.lang.Object

void	notify() Wakes up a single thread that is waiting on this object's monitor.
void	notifyAll() Wakes up all threads that are waiting on this object's monitor.
void	wait() Causes the current thread to wait until another thread invokes the notify() method or the notifyAll() method for this object.
void	wait(long timeout) Causes the current thread to wait until either another thread invokes the notify() method or the notifyAll() method for this object, or a specified amount of time has elapsed.
void	wait(long timeout, int nanos) Causes the current thread to wait until another thread invokes the notify() method or the notifyAll() method for this object, or some other thread interrupts the current thread, or a certain amount of real time has elapsed.

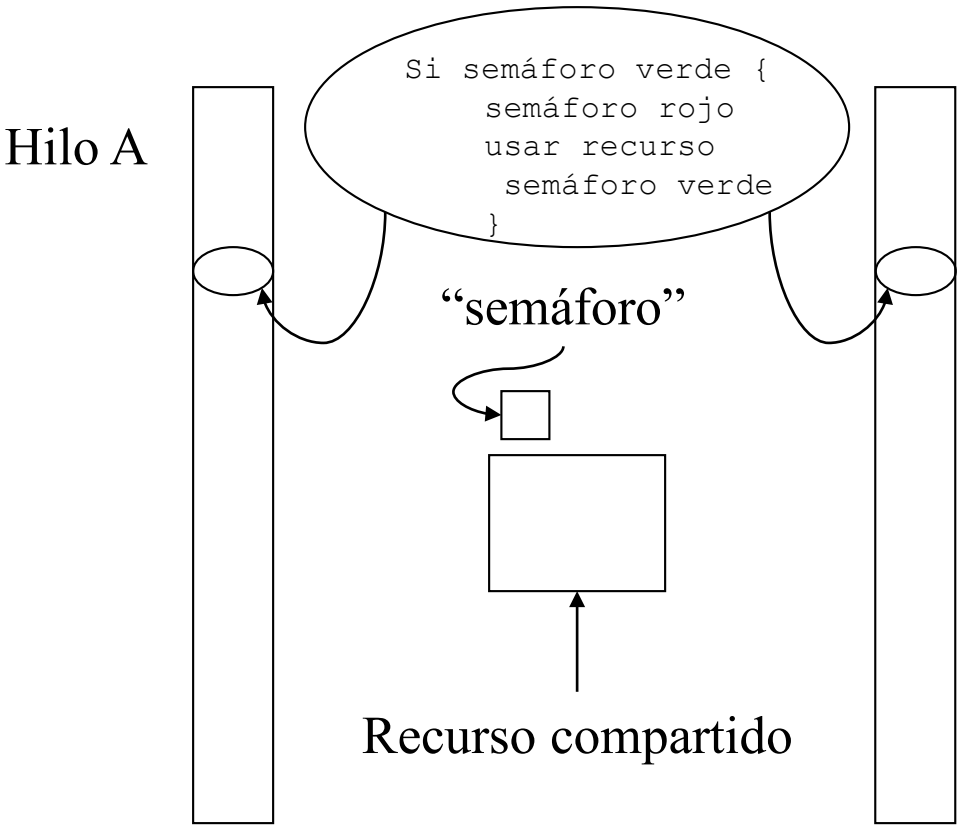
java.lang.Thread

static void	sleep(long millis) Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds, subject to the precision and accuracy of system timers and schedulers.
static void	sleep(long millis, int nanos) Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds plus the specified number of nanoseconds, subject to the precision and accuracy of system timers and schedulers.
void	start() Causes this thread to begin execution; the Java Virtual Machine calls the run method of this thread.
void	stop() **Deprecated.** *This method is inherently unsafe. Stopping a thread with Thread.stop causes it to unlock all of the monitors that it has locked (as a natural consequence of the unchecked ThreadDeath exception propagating up the stack). If any of the objects previously protected by these monitors were in an inconsistent state, the damaged objects become visible to other threads, potentially resulting in arbitrary behavior. Many uses of stop should be replaced by code that simply modifies some variable to indicate that the target thread should stop running. The target thread should check this variable regularly, and return from its run method in an orderly fashion if the variable indicates that it is to stop running. If the target thread waits for long periods (on a condition variable, for example), the interrupt method should be used to interrupt the wait. For more information, see Why are Thread.stop, Thread.suspend and Thread.resume Deprecated?.*
void	stop(Throwable obj) **Deprecated.** *This method is inherently unsafe. See stop() for details. An additional danger of this method is that it may be used to generate exceptions that the target thread is unprepared to handle (including checked exceptions that the thread could not possibly throw, were it not for this method). For more information, see Why are Thread.stop, Thread.suspend and Thread.resume Deprecated?.*
void	suspend() **Deprecated.** *This method has been deprecated, as it is inherently deadlock-prone. If the target thread holds a lock on the monitor protecting a critical system resource when it is suspended, no thread can access this resource until the target thread is resumed. If the thread that would resume the target thread attempts to lock this monitor prior to calling resume, deadlock results. Such deadlocks typically manifest themselves as "frozen" processes. For more information, see Why are Thread.stop, Thread.suspend and Thread.resume Deprecated?.*
String	toString() Returns a string representation of this thread, including the thread's name, priority, and thread group.
static void	yield() A hint to the scheduler that the current thread is willing to yield its current use of a processor.

Mecanismos proporcionados por Java para el entorno multi-hilo

- Exclusión mutua (secciones criticas)
- Bloqueo de recursos

Palabras reservadas en Java				
abstract	assert***	boolean	break	byte
case	catch	char	class	const*
continue	default	do	double	else
enum***	extends	final	finally	float
for	goto*	if	implements	import
instanceof	int	interface	long	native
new	package	private	protected	public
return	short	static	strictfp**	super
switch	synchronized	this	throw	throws
transient	try	void	volatile	while



Interbloqueo (deadlock)

Es una forma "terminal" de inanición. Ocurre cuando dos o más hilos esperan a una condición que no puede satisfacerse. El interbloqueo más habitual consiste en que dos (o más) hilos esperan a que otro haga algo de un modo circular.

```
public class Cubiculo {
            private int contenido;
            private boolean disponible = false;

            public synchronized int get() {
                ...
            }

            public synchronized void put(int valor) {
                ...
            }
        }
```

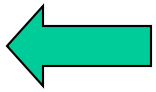
Sección crítica

wait / notify

```
public synchronized int get() {
  while (!disponible) {
   // esperar a que el productor genere un valor
    try { wait();} catch (InterruptedException e) {}
  }
  disponible = false;
  // notificar al productor que el valor ha sido recogido
  notifyAll();
  return contenido;
  }

public synchronized void put(int valor) {
  while (disponible) {
    // esperar a que el consumidor recoja un valor
    try { wait(); } catch (InterruptedException e) {}
  }
  contenido = valor;
  disponible = true;
  // notificar al consumidor que el valor ha sido generado
  notifyAll();
}
```

Productor / consumidor



```
public class Productor extends Thread {
            private Cubiculo cubiculo;
            private int numero;

            public Productor(Cubiculo c, int numero) {
                cubiculo = c; this.numero = numero;
            }

            public void run() {
                for (int i = 0; i < 10; i++) {
                    cubiculo.put(i);
                    System.out.println("("+numero+ ") >> " + i);
                    try {
                        sleep((int)(Math.random() * 100));
                    } catch (InterruptedException e) { }
                }
            }
        }
```

```
public class Consumidor extends Thread {
    private Cubiculo cubiculo;
    private int numero;

    public Consumidor(Cubiculo c, int numero) {
        cubiculo = c; this.numero = numero;
        setDaemon(true);
    }

    public void run() {
        int valor = 0;
        while (true) {
            valor = cubiculo.get();
            System.out.println("   (" + numero+ ") << " + valor);
            yield();
        }
    }
}
```

Comprobando el funcionamiento

```
public class MainProdCons extends Object {

  public static void main (String args[]) {
    Cubiculo cubiculo=new Cubiculo();
    Productor p1=new Productor(cubiculo,1);
    Productor p2=new Productor(cubiculo,2);
    Productor p3=new Productor(cubiculo,3);
    Consumidor c1=new Consumidor(cubiculo,1);
    Consumidor c2=new Consumidor(cubiculo,2);
    Consumidor c3=new Consumidor(cubiculo,3);

    p1.start();
    p2.start();
    p3.start();
    c1.start();
    c2.start();
    c3.start();

    }

}
```

```
(1) >> 0
  (1) << 0
(2) >> 0
  (2) << 0
(3) >> 0
  (3) << 0
(2) >> 1
  (1) << 1
(3) >> 1
  (2) << 1
(1) >> 1
  (3) << 1
(1) >> 2
  (1) << 2
(3) >> 2
  (2) << 2
(2) >> 2
  (3) << 2
(3) >> 3
  (1) << 3
(1) >> 3
  (2) << 3
(3) >> 4
  (3) << 4
(2) >> 3
  (1) << 3
(1) >> 4
  (2) << 4
(2) >> 4
  (3) << 4
```

```
(3) >> 5
  (1) << 5
(2) >> 5
  (2) << 5
(3) >> 6
  (3) << 6
(1) >> 5
  (1) << 5
(2) >> 6
  (2) << 6
(3) >> 7
  (3) << 7
(2) >> 7
  (1) << 7
(1) >> 6
  (2) << 6
(3) >> 8
  (3) << 8
(2) >> 8
(1) >> 7
  (1) << 8
  (2) << 7
(3) >> 9
  (3) << 9
(2) >> 9
  (1) << 9
(1) >> 8
  (2) << 8
(1) >> 9
  (3) << 9
```

Ojo!. Algo va mal

Arreglado... (no todo)

```
public void run() {
  for (int i = 0; i < 10; i++) {
    synchronized(cubiculo){
      cubiculo.put(i);
      System.out.println("("+numero+ ") >> " + i);
    }
    try {
      sleep((int)(Math.random() * 100));
    } catch (InterruptedException e) { }
  }
}
```

Productor / consumidor

```
public void run() {
  int valor = 0;
  while (true) {
    synchronized(cubiculo) {
      valor = cubiculo.get();
      System.out.println("   (" + numero+ ") << " + valor);
    }
    yield();
  }
}
```

```
(1) >> 0
  (1) << 0
(1) >> 1
  (2) << 1
(2) >> 0
  (3) << 0
(3) >> 0
  (1) << 0
(1) >> 2
  (2) << 2
(2) >> 1
  (3) << 1
(3) >> 1
  (1) << 1
(1) >> 3
  (2) << 3
(2) >> 2
  (3) << 2
(3) >> 2
  (1) << 2
(2) >> 3
  (2) << 3
(1) >> 4
  (3) << 4
(3) >> 3
  (1) << 3
(2) >> 4
  (2) << 4
(1) >> 5
  (1) << 5
```

```
(3) >> 4
  (2) << 4
(2) >> 5
  (3) << 5
(1) >> 6
  (1) << 6
(3) >> 5
  (2) << 5
(2) >> 6
  (3) << 6
(1) >> 7
  (1) << 7
(3) >> 6
  (2) << 6
(2) >> 7
  (3) << 7
(1) >> 8
  (1) << 8
(3) >> 7
  (2) << 7
(2) >> 8
  (1) << 8
(1) >> 9
  (3) << 9
(3) >> 8
  (2) << 8
(2) >> 9
  (1) << 9
(3) >> 9
```

Ojo!. Algo va mal

La solución definitiva?

(ThreadGroup)

```
public class MainProdCons extends Object {

public static void main (String args[]) {
    Cubiculo cubiculo=new Cubiculo();

    ThreadGroup productores=new ThreadGroup("productores");
    ThreadGroup consumidores=new ThreadGroup("consumidores");

    Productor p1=new Productor(productores,cubiculo,"1"); p1.start();
    Productor p2=new Productor(productores,cubiculo,"2"); p2.start();
    Productor p3=new Productor(productores,cubiculo,"3"); p3.start();
    Consumidor c1=new Consumidor(consumidores,cubiculo,"1"); c1.start();
    Consumidor c2=new Consumidor(consumidores,cubiculo,"2"); c2.start();
    Consumidor c3=new Consumidor(consumidores,cubiculo,"3"); c3.start();

    consumidores.setDaemon(true);

    int n;
    while ((n=productores.activeCount())!=0) {
        System.out.println("Productores Activos= "+n);
        try { Thread.sleep(500); } catch (InterruptedException e) {}
    }
    try { Thread.sleep(100); } catch (InterruptedException e) {}
    System.out.println("Productores Activos= "+n);
    }
}
```

```
Productores Activos= 3
(1) >> 0
  (1) << 0
(3) >> 0
  (2) << 0
(1) >> 1
  (3) << 1
(1) >> 2
  (1) << 2
(2) >> 0
  (2) << 0
(3) >> 1
  (3) << 1
```

```
(2) >> 1
  (1) << 1
(1) >> 3
  (2) << 3
(2) >> 2
  (3) << 2
(3) >> 2
  (1) << 2
(1) >> 4
  (2) << 4
(3) >> 3
  (3) << 3
(2) >> 3
```

```
  (1) << 3
(3) >> 4
  (2) << 4
Productores Activos= 3
(1) >> 5
  (3) << 5
(2) >> 4
  (1) << 4
(1) >> 6
  (2) << 6
(3) >> 5
  (3) << 5
(2) >> 5
```

```
    (1) << 5
(1) >> 7
  (2) << 7
(2) >> 6
  (3) << 6
(3) >> 6
  (1) << 6
(1) >> 8
Productores Activos= 3
  (2) << 8
(2) >> 7
  (3) << 7
(3) >> 7
  (1) << 7
(1) >> 9
  (2) << 9
(2) >> 8
  (3) << 8
(3) >> 8
  (1) << 8
(2) >> 9
  (2) << 9
(3) >> 9
  (3) << 9
Productores Activos= 0
```

```
public class Productor extends Thread {
            private Cubiculo cubiculo;

            public Productor(ThreadGroup tg, Cubiculo c, String id) {
                super(tg,id); cubiculo = c;
            }

            public void run() {
                for (int i = 0; i < 10; i++) {
                   synchronized(cubiculo){cubiculo.put(i);
                                          System.out.println("("+getName()+ ") >> " + i);
                                         }
                    try {sleep((int)(Math.random() * 100));
                    } catch (InterruptedException e) { }
                }
            }
        }
```

Productor / consumidor

```
public class Consumidor extends Thread {
    private Cubiculo cubiculo;

    public Consumidor(ThreadGroup tg, Cubiculo c, String id) {
        super(tg,id); cubiculo = c; setDaemon(true);
    }

    public void run() {
        int valor = 0;
        while (true) {
            synchronized(cubiculo) {valor = cubiculo.get();
                                     System.out.println("  (" + getName()+ ") << " + valor);
                                    }
            yield();
        }
    }
}
```