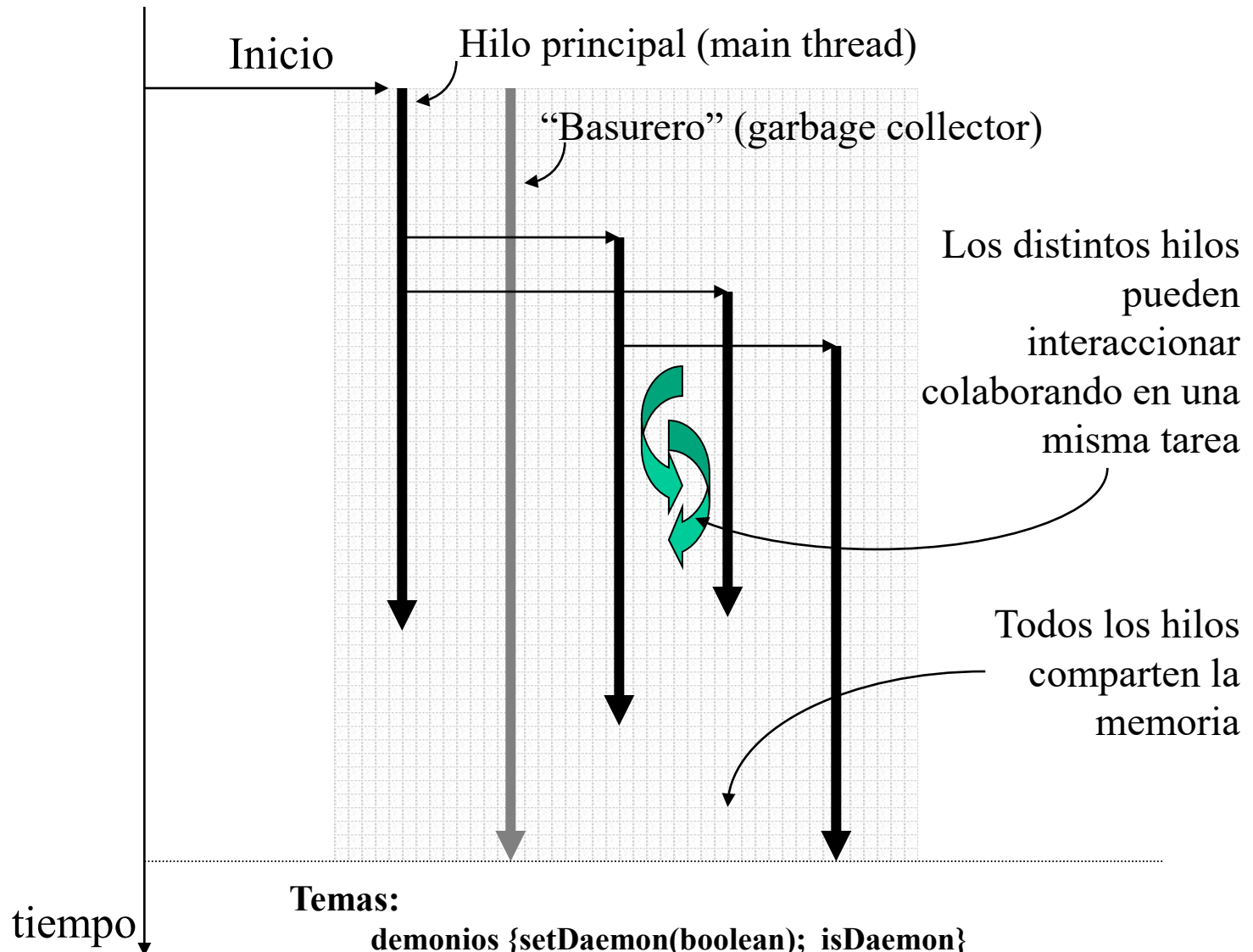


Threads (Hilos)

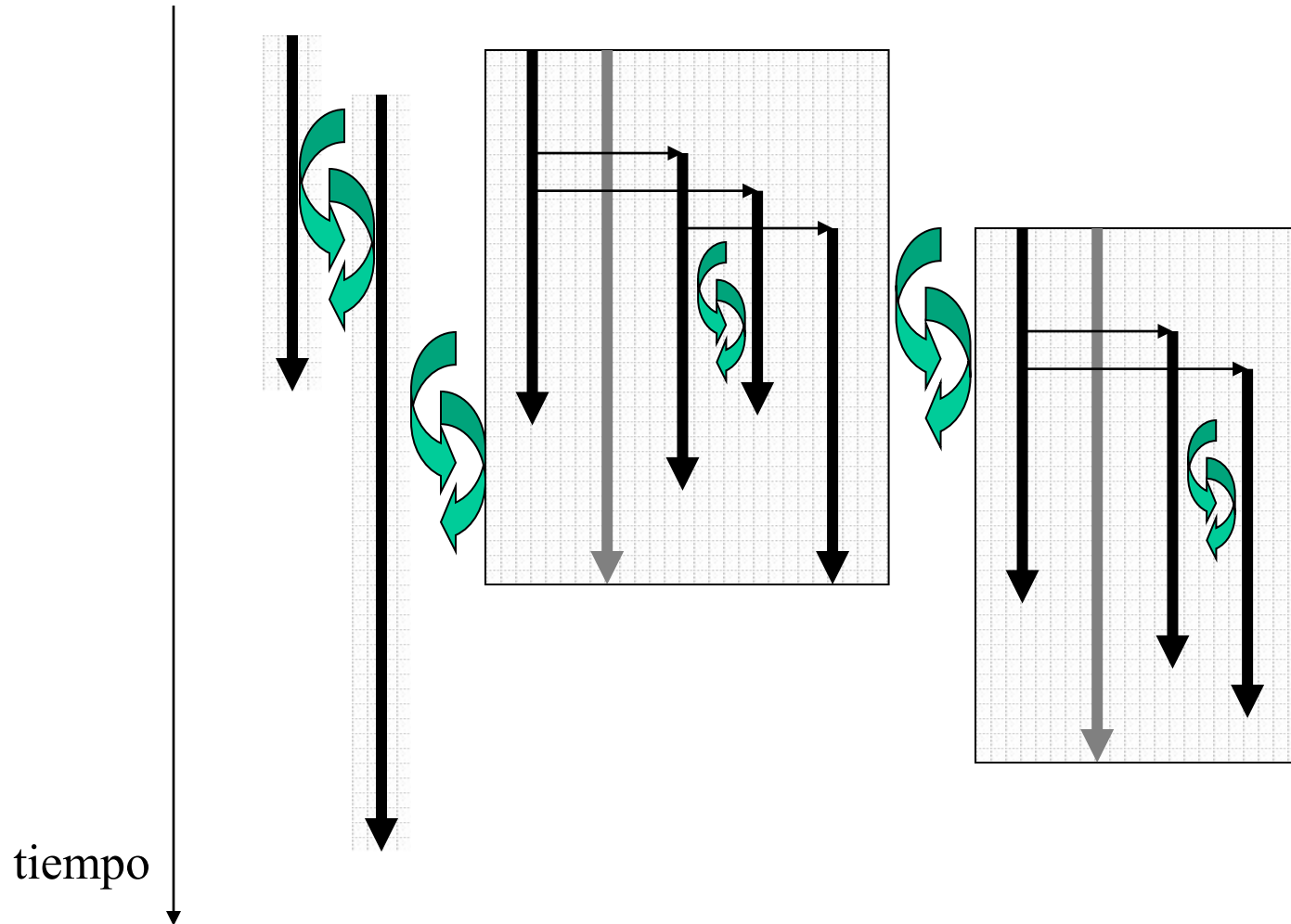


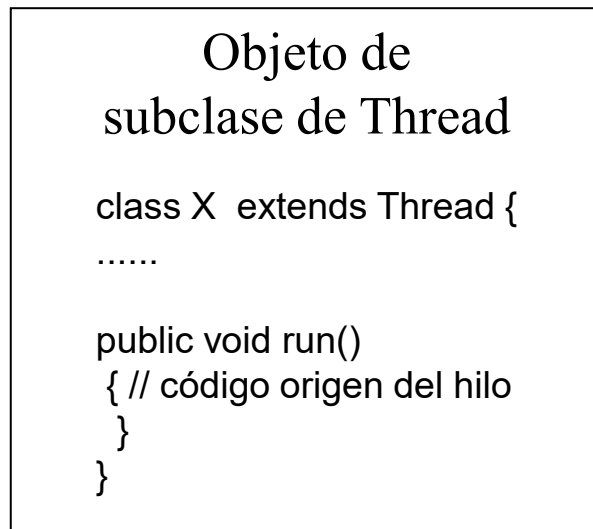
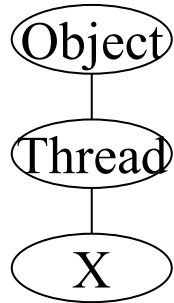
Temas:

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



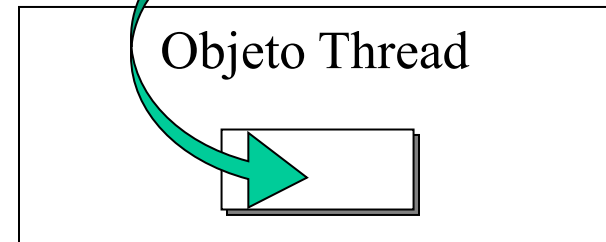
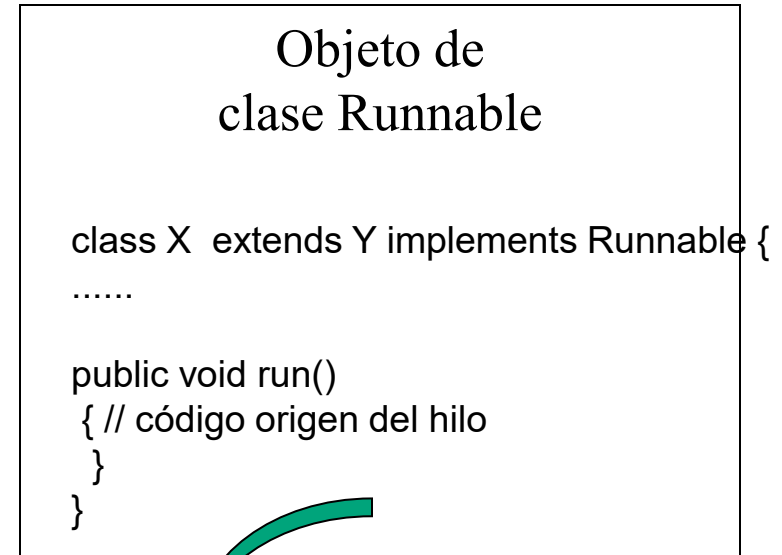
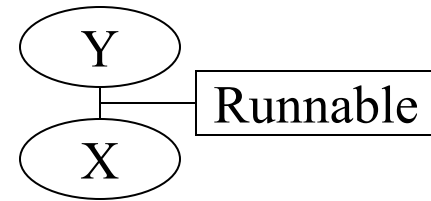
Procesos en un S.O.





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

*El start() puede situarse en el constructor

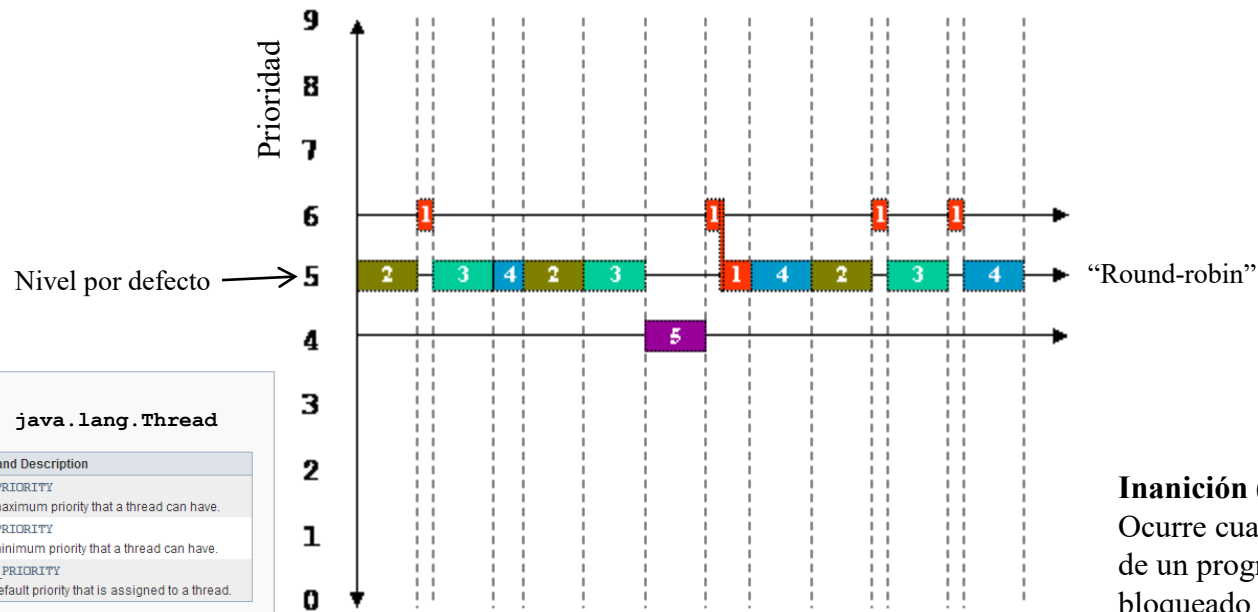


X a = new X(); Thread t=new Thread(a); t.start();



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`

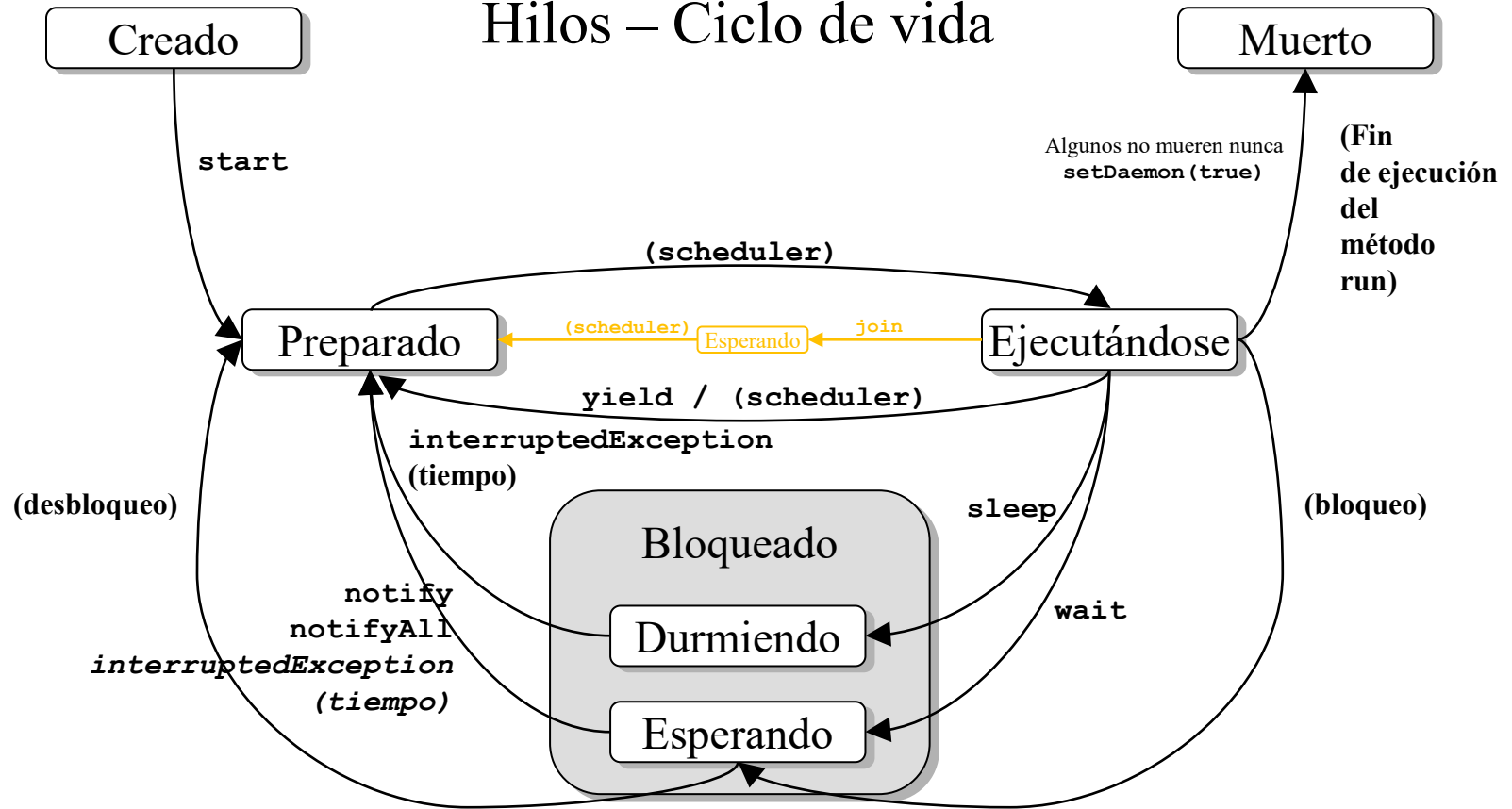
```
•public final void setPriority(int newPriority)
•public final int getPriority()
```

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	<code>notify()</code>	Wakes up a single thread that is waiting on this object's monitor.
void	<code>notifyAll()</code>	Wakes up all threads that are waiting on this object's monitor.
void	<code>wait()</code>	Causes the current thread to wait until another thread invokes the <code>notify()</code> method or the <code>notifyAll()</code> method for this object.
void	<code>wait(long timeout)</code>	Causes the current thread to wait until either another thread invokes the <code>notify()</code> method or the <code>notifyAll()</code> method for this object, or a specified amount of time has elapsed.
void	<code>wait(long timeout, int nanos)</code>	Causes the current thread to wait until another thread invokes the <code>notify()</code> method or the <code>notifyAll()</code> 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	<code>sleep(long millis)</code>	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	<code>sleep(long millis, int nanos)</code>	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	<code>start()</code>	Causes this thread to begin execution; the Java Virtual Machine calls the <code>run()</code> method of this thread.
void	<code>stop()</code>	Deprecated. This method is inherently unsafe. Stopping a thread with <code>Thread.stop</code> causes it to unlock all of the monitors that it has locked (as a natural consequence of the unchecked <code>ThreadDeath</code> 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 <code>stop</code> 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 <code>run</code> 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 <code>interrupt</code> method should be used to interrupt the wait. For more information, see <i>Why are <code>Thread.stop</code>, <code>Thread.suspend</code> and <code>Thread.resume</code> Deprecated?</i>
void	<code>stop(Throwable obj)</code>	Deprecated. This method is inherently unsafe. See <i>stop()</i> 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 <i>Why are <code>Thread.stop</code>, <code>Thread.suspend</code> and <code>Thread.resume</code> Deprecated?</i>
void	<code>suspend()</code>	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 <code>resume</code> , deadlock results. Such deadlocks typically manifest themselves as "frozen" processes. For more information, see <i>Why are <code>Thread.stop</code>, <code>Thread.suspend</code> and <code>Thread.resume</code> Deprecated?</i>
String	<code>toString()</code>	Returns a string representation of this thread, including the thread's name, priority, and thread group.
static void	<code>yield()</code>	A hint to the scheduler that the current thread is willing to yield its current use of a processor.

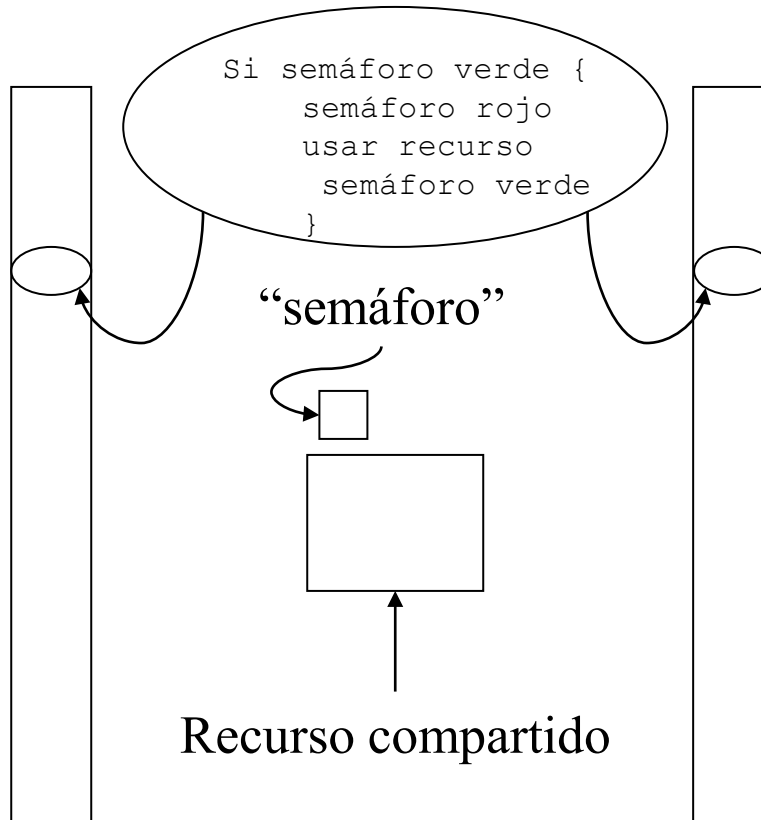


Mecanismos proporcionados por Java para el entorno multi-hilo

- Exclusión mutua (secciones críticas)
- 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

Hilo A



Hilo B

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.

Sincronizando un contador

```
package edu.upvehu.gbg.docencia.sincronizacion;

import java.util.concurrent.atomic.AtomicInteger;

public class ContandoConHilos {
    private static final int TOP_COUNT=1_000_000;

    public static void main(String[] args) throws InterruptedException {

        //Los dos hilos "contra" un contador no Thread Safe
        System.out.printf("Contador no Thread Safe: %,d%n",
            lanzaEjecucionADosHilos(new NoTSCounter(), TOP_COUNT));

        //Los dos hilos "contra" un contador Thread Safe mediante sincronización
        System.out.printf("Contador Thread Safe: %,d%n",
            lanzaEjecucionADosHilos(new synchronizedCounter(), TOP_COUNT));

        //Los dos hilos "contra" un contador volatil mediante sincronización
        System.out.printf("Contador volatil: %,d%n",
            lanzaEjecucionADosHilos(new volatileCounter(), TOP_COUNT));

        //Los dos hilos "contra" un contador atómico
        System.out.printf("Contador atómico: %,d%n",
            lanzaEjecucionADosHilos(new AtomicCounter(), TOP_COUNT));
    }

    private static int lanzaEjecucionADosHilos(Counter counter, int topCount)
        throws InterruptedException {
        Thread t1=new Barrido(counter, topCount), t2=new Barrido(counter, topCount);
        t1.start(); t2.start(); t1.join(); t2.join();
        return counter.getCount();
    }
}
```

```
//Hilo que "barre" un contador de cero hasta un máximo
class Barrido extends Thread{
    Counter counter; int topCount;

    public Barrido(Counter counter, int topCount) {this.counter = counter; this.topCount=topCount; }

    @Override public void run() { for (int i = 0; i < topCount; i++) counter.increment();}
}
```

```
//Interfaz del concepto "contador"
interface Counter {
    public void increment();
    public int getCount();
}
```

```
//Contador no Thread Safe
class NoTSCounter implements Counter {
    private int count = 0;

    @Override public void increment() { count++; }
    @Override public int getCount() { return count; }
}
```

```
//Contador Thread Safe
class synchronizedCounter implements Counter {
    private int count = 0;

    @Override public synchronized void increment() { count++; }
    @Override public synchronized int getCount() { return count; }
}
```

```
//Contador volatil (no es TS porque count++ no es atómica)
class volatileCounter implements Counter {
    private volatile int count = 0;

    @Override public void increment() { count++; }
    @Override public int getCount() { return count; }
}
```

```
//Contador atómico (es TS —la atomicidad lo asegura)
class AtomicCounter implements Counter {
    private final AtomicInteger count = new AtomicInteger(0);

    @Override public void increment() { count.incrementAndGet(); }
    @Override public int getCount() { return count.get(); }
}
```

```
german@Raskolnikov:Java -jar ContandoConHilos.jar
Contador no Thread Safe: 1.060.434 ← Carreras críticas y problemas de visibilidad
Contador Thread Safe: 2.000.000
Contador volatil: 1.220.971 ← Carreras críticas
Contador atómico: 2.000.000
```



Desarrollo de un ejemplo con hilos en colaboración

Hilos productores de datos



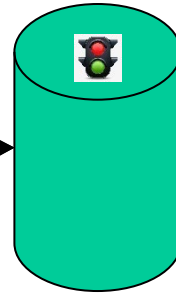
Simularemos los productores como generadores de una serie enteros (de 0 a 9) en intervalos de tiempo aleatorios.

Hilos consumidores de datos



Simularemos los consumidores únicamente para recoger los datos. Estarán siempre dispuestos a recoger un dato.

Recurso compartido buffer sincronizado



Simularemos el buffer de la manera más simple posible: con capacidad para un solo entero. El semáforo será booleano)

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

    public synchronized int get() {
        ...
    }

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

Sección crítica

Un método sincronizado es una “sección crítica” de código. Toma en propiedad el objeto, de modo que otro hilo no puede acceder simultáneamente a cualquier otro método sincronizado.

(es vital que los campos “críticos” sean privados y sólo accedidos desde métodos sincronizados)

wait / notify

Un hilo puede esperar a que algo suceda mediante un “wait”, y retomar su ejecución cuando sea notificado (“notify”)

En nuestro ejemplo se notifica a todos los hilos (notifyAll) porque así eliminamos la posibilidad de que al pasar a preparado sólo uno resulte ser del mismo tipo (consumidor → consumidor, o productor → productor)

```
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;
}

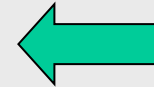
public synchronized void put(int valor) {
    while (disponible) {
        // esperar a que el consumidor recoja un valor
        try { wait(); } catch (InterruptedException e) {}
    }
    contenido = valor;
    disponible = true;
    // notificar al consumidor que el valor ha sido generado
    notifyAll();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



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

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

Productor / consumidor



```
public void run() {
    for (int i = 0; i < 10; i++) {
        cubiculo.put(i);
        System.out.println("(" + numero + ") >> " + i);
        //Usamos un sleep en un ciclo por claridad, pero es mala práctica
        try {
            Thread.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);
            Thread.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();  
    }  
}
```

Ojo!. Algo va mal

(1) >> 0	(3) >> 5
(1) << 0	(1) << 5
(2) >> 0	(2) >> 5
(2) << 0	(2) << 5
(3) >> 0	(3) >> 6
(3) << 0	(3) << 6
(2) >> 1	(1) >> 5
(1) << 1	(1) << 5
(3) >> 1	(2) >> 6
(2) << 1	(2) << 6
(1) >> 1	(3) >> 7
(3) << 1	(3) << 7
(1) >> 2	(2) >> 7
(1) << 2	(1) << 7
(3) >> 2	(1) >> 6
(2) << 2	(2) << 6
(2) >> 2	(3) >> 8
(3) << 2	(3) << 8
(3) >> 3	(2) >> 8
(1) << 3	(1) >> 7
(1) >> 3	(1) << 8
(2) << 3	(2) << 7
(3) >> 4	(3) >> 9
(3) << 4	(3) << 9
(2) >> 3	(2) >> 9
(1) << 3	(1) << 9
(1) >> 4	(1) >> 8
(2) << 4	(2) << 8
(2) >> 4	(1) >> 9
(3) << 4	(3) << 9



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();  
    }  
}
```

Ojo!. Algo va mal

(1) >> 0	(3) >> 4
(1) << 0	(2) << 4
(1) >> 1	(2) >> 5
(2) << 1	(3) << 5
(2) >> 0	(1) >> 6
(3) << 0	(1) << 6
(3) >> 0	(3) >> 5
(1) << 0	(2) << 5
(1) >> 2	(2) >> 6
(2) << 2	(3) << 6
(2) >> 1	(1) >> 7
(3) << 1	(1) << 7
(3) >> 1	(3) >> 6
(1) << 1	(2) << 6
(1) >> 3	(2) >> 7
(2) << 3	(3) << 7
(2) >> 2	(1) >> 8
(3) << 2	(1) << 8
(3) >> 2	(3) >> 7
(1) << 2	(2) << 7
(2) >> 3	(2) >> 8
(2) << 3	(1) << 8
(1) >> 4	(1) >> 9
(3) << 4	(3) << 9
(3) >> 3	(3) >> 8
(1) << 3	(2) << 8
(2) >> 4	(2) >> 9
(2) << 4	(1) << 9
(1) >> 5	(3) >> 9
(1) << 5	



```
public class MainProdCons {
```

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

```
        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);
    }
```

```
}
```

La solución definitiva?

Esperar en main() a la terminación de los productores con join() y dar un tiempo extra.

Aprovechamos para introducir la existencia de **ThreadGroup**

En vez de usar join(), monitorizaremos la finalización del grupo.

```

        (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
```

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
```



```

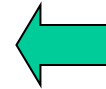
public class Productor extends Thread {
    private Cubiculo cubiculo;

    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();
        }
    }
}

```

