Programación multihilo

Recursos compartidos por los hilos.

Cuando creamos varios objetos de una clase, puede ocurrir que varios hilos de ejecución accedan a un objeto. Es importante recordar que **todos los campos del objeto son compartidos entre todos los hilos**.

Supongamos una clase como esta:

```
public class Empleado(){
        int numHorasTrabajadas=0;
        public void int incrementarHoras(){
                numHorasTrabajadas++;
        }
}
```

Si varios hilos ejecutan sin querer el método `incrementar` en teoría debería ocurrir que el número se incrementase tantas veces como procesos. Sin embargo, **es muy probable que eso no ocurra**

Estados de un hilo. Cambios de estado. ¶

Aunque no lo vemos, un hilo cambia de estado: puede pasar de la nada a la ejecución. De la ejecución al estado «en espera». De ahí puede volver a estar en ejecución. De cualquier estado se puede pasar al estado «finalizado».

El programador no necesita controlar esto, lo hace el sistema operativo. Sin embargo un programa multihilo mal hecho puede dar lugar problemas como los siguientes:

- Interbloqueo. Se produce cuando las peticiones y las esperas se entrelazan de forma que ningún proceso puede avanzar.
- Inanición. Ningún proceso consigue hacer ninguna tarea útil y por lo tanto hay que esperar a que el administrador del sistema detecte el interbloqueo y mate procesos (o hasta que alguien reinicie el equipo).

Elementos relacionados con la programación de hilos. Librerías y clases.

Para crear programas multihilo en Java se pueden hacer dos cosas:

1. Heredar de la clase Thread.
2. Implementar la interfaz Runnable.

Los documentos de Java aconsejan el segundo. Lo único que hay que hacer es algo como esto.

```
class EjecutorTareaCompleja implements Runnable{
        private String nombre;
        int numEjecucion;
        public EjecutorTareaCompleja(String nombre){
                this.nombre=nombre;
        }
        @Override
        public void run() {
                String cad;
                while (numEjecucion<100){
                        for (double i=0; i<4999.99; i=i+0.04)
                        {
                                Math.sqrt(i);
                        }
```

```
                        cad="Soy el hilo "+this.nombre;
                        cad+=" y mi valor de i es "+numEjecucion;
                        System.out.println(cad);
                        numEjecucion++;
                }
        }

}
public class LanzaHilos {

        /**
         * @param args
         */
        public static void main(String[] args) {
                int NUM_HILOS=500;
                EjecutorTareaCompleja op;
                for (int i=0; i<NUM_HILOS; i++)
                {
                        op=new EjecutorTareaCompleja ("Operacion "+i);
                        Thread hilo=new Thread(op);
                        hilo.start();
                }
        }
}
```

Advertencia:

Este código tiene un problema **muy grave** y es que no se controla el acceso a variables compartidas, es decir **HAY UNA SECCIÓN CRÍTICA QUE NO ESTÁ PROTEGIDA** por lo que el resultado de la ejecución no muestra ningún sentido aunque el programa esté bien.

Gestión de hilos.

Con los hilos se pueden efectuar diversas operaciones que sean de utilidad al programador (y al administrador de sistemas a veces).

Por ejemplo, un hilo puede tener un nombre. Si queremos asignar un nombre a un hilo podemos usar el método `setName("Nombre que sea")`. También podemos obtener un objeto que represente el hilo de ejecución con `currentThread` que nos devolverá un objeto de la clase `Thread`.

Otra operación de utilidad al gestionar hilos es indicar la prioridad que queremos darle a un hilo. En realidad esta prioridad es indicativa, el sistema operativo no está obligado a respetarla aunque por lo general lo hacen. Se puede indicar la prioridad con `setPriority(10)`. La máxima prioridad posible es `MAX_PRIORITY`, y la mínima es `MIN_PRIORITY`.

Cuando lanzamos una operación también podemos usar el método `Thread.sleep(numero)` y poner nuestro hilo «a dormir».

Cuando se trabaja con prioridades en hilos **no hay garantías de que un hilo termine cuando esperemos**.

Podemos terminar un hilo de ejecución llamando al método `join`. Este método devuelve el control al hilo principal que lanzó el hilo secundario con la posibilidad de elegir un tiempo de espera en milisegundos.

El siguiente programa ilustra el uso de estos métodos:

```
class Calculador implements Runnable{
        @Override
        public void run() {
                int num=0;
                while(num<5){
                        System.out.println("Calculando...");
```

```
                        try {
                                long tiempo=(long) (1000*Math.random()*10);
                                if (tiempo>8000){
                                        Thread hilo=Thread.currentThread();
                                        System.out.println(
                                                        "Terminando hilo:"+
                                                                        hilo.getName()
                                        );
                                        hilo.join();
                                }
                                Thread.sleep(tiempo);
                        } catch (InterruptedException e) {
                                // TODO Auto-generated catch block
                                e.printStackTrace();
                        }
                        System.out.println("Calculado y reiniciando.");
                        num++;
                }
                Thread hilo=Thread.currentThread();
                String miNombre=hilo.getName();
                System.out.println("Hilo terminado:"+miNombre);
        }
}

public class LanzadorHilos {
        public static void main(String[] args) {
                Calculador vHilos[]=new Calculador[5];
                for (int i=0; i<5;i++){
                        vHilos[i]=new Calculador();
                        Thread hilo=new Thread(vHilos[i]);
                        hilo.setName("Hilo "+i);
                        if (i==0){
                                hilo.setPriority(Thread.MAX_PRIORITY);
                        }
                        hilo.start();
                }
        }
}
```

Ejercicio: crear un programa que lance 10 hilos de ejecución donde a cada hilo se le pasará la base y la altura de un triángulo, y cada hilo ejecutará el cálculo del área de dicho triángulo informando de qué base y qué altura recibió y cual es el área resultado.

Una posibilidad (quizá incorrecta) sería esta:

Para implementar la clase `CalculadorAreas`

```
class CalculadorAreas implements Runnable {

    float base, altura, area;

    public int contador = 0;

    public CalculadorAreas(float b, float a) {
        this.base = b;
        this.altura = a;
    }

    public synchronized void incrementarContador() {
        contador = contador + 1;
    }

    @Override
    public void run() {
        Random generador;
        generador = new Random();
        area = base * altura / 2;
```

```
            try {
                Thread.sleep(generador.nextInt(650));
            } catch (InterruptedException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }
            this.incrementarContador();
        }
    }
```

Para implementar la clase `AreasEnParalelo` que lanza los hilos para hacer muchos cálculos de áreas en paralelo usaremos esto:

```
public class AreasEnParalelo {

    public static void main(String[] args) throws
    InterruptedException {
        CalculadorAreas ca = new CalculadorAreas(1, 2);
        final int MAX_HILOS = 10000;
        Thread[] hilos = new Thread[MAX_HILOS];
        for (int i = 0; i < MAX_HILOS; i++) {
            hilos[i] = new Thread(ca);
            hilos[i].start();
        }
        for (int i = 0; i < MAX_HILOS; i++) {
            hilos[i].join();
        }
        System.out.println("Total de calculos:" + ca.contador);
    }
}
```

```
package com.ies;

import java.util.Random;

class CalculadorAreas implements Runnable{
        int base, altura;
        public CalculadorAreas(int base, int altura){
                this.base=base;
                this.altura=altura;
        }
        @Override
        public void run() {
                float area=this.base*this.altura/2;
                System.out.print("Base:"+this.base);
                System.out.print("Altura:"+this.altura);
                System.out.println("Area:"+area);
        }

}
public class AreasEnParalelo {

        public static void main(String[] args) {
                Random generador=new Random();
                int numHilos=10000;
                int baseMaxima=3;
                int alturaMaxima=5;
                for (int i=0; i<numHilos; i++){
                        //Sumamos 1 para evitar casos como base=0
                        int base=1+generador.nextInt(baseMaxima);
                        int altura=1+generador.nextInt(alturaMaxima);
                        CalculadorAreas ca=
                                        new CalculadorAreas(base, altura);
                        Thread hiloAsociado=new Thread(ca);
                        hiloAsociado.start();
                }
```

```
        }
    }
```

Las secciones siguientes sirven como resumen de como crear una aplicación multihilo

Creación, inicio y finalización.

- Podemos heredar de Thread o implementar `Runnable`. Usaremos el segundo recordando implementar el método `public void run()`.
- Para crear un hilo asociado a un objeto usaremos algo como:

```
Thread hilo=new Thread(objetoDeClase)
```

Lo más habitual es guardar en un vector todos los hilos que hagan algo, y no en un objeto suelto.

- Cada objeto que tenga un hilo asociado debe iniciarse así:

```
hilo.start();
```

- Todo programa multihilo tiene un «hilo principal», el cual deberá esperar a que terminen los hilos asociados ejecutando el método `join()` .

Sincronización de hilos.

Cuando un método acceda a una variable miembro que esté compartida deberemos proteger dicha sección crítica, usando `synchronized`. Se puede poner todo el método `synchronized` o marcar un trozo de código más pequeño.

Información entre hilos.

Todos los hilos comparten todo, así que obtener información es tan sencillo como consultar un miembro. En realidad podemos comunicar los hilos con otro mecanismo llamado `sockets de red`, pero se ve en el tema siguiente.

Prioridades de los hilos.

Podemos asignar distintas prioridades a los hilos usando los campos estáticos `MAX_PRIORITY` y `MIN_PRIORITY`. Usando valores entre estas dos constantes podemos hacer que un hilo reciba más procesador que otro (se hace en contadas ocasiones).

Para ello se usa el método `setPriority(valor)`

Gestión de prioridades.

En realidad un sistema operativo no está obligado a respetar las prioridades, sino que se lo tomará como «recomendaciones». En general hasta ahora todos respetan hasta cierto punto las prioridades que pone el programador pero no debe tomarse como algo absoluto.

Programación de aplicaciones multihilo.

La estructura típica de un programa multihilo es esta:

```
class TareaCompleja implements Runnable{
        @Override
        public void run() {
                for (int i=0; i<100;i++){
```

```
                        int a=i*3;
                }
                Thread hiloActual=Thread.currentThread();
                String miNombre=hiloActual.getName();
                System.out.println(
                                "Finalizado el hilo"+miNombre);
        }
}
public class LanzadorHilos {
        public static void main(String[] args) {
                int NUM_HILOS=100;
                Thread[] hilosAsociados;

                hilosAsociados=new Thread[NUM_HILOS];
                for (int i=0;i<NUM_HILOS;i++){
                        TareaCompleja t=new TareaCompleja();
                        Thread hilo=new Thread(t);
                        hilo.setName("Hilo: "+i);
                        hilo.start();
                        hilosAsociados[i]=hilo;
                }

                /* Despues de crear todo, nos aseguramos
                 * de esperar que todos los hilos acaben. */

                for (int i=0; i<NUM_HILOS; i++){
                        Thread hilo=hilosAsociados[i];
                        try {
                                //Espera a que el hilo acabe
                                hilo.join();
                        } catch (InterruptedException e) {
                                System.out.print("Algun hilo acabó ");
                                System.out.println(" antes de tiempo!");
                        }
                }
                System.out.println("El principal ha terminado");
        }
}
```

Supongamos que la tarea es más compleja y que el bucle se ejecuta un número al azar de veces. Esto significaría que nuestro bucle es algo como esto:

```
Random generador= new Random();
int numAzar=(1+generador.nextInt(5))*100;
for (int i=0; i<numAzar;i++){
        int a=i*3;
}
```

¿Como podríamos modificar el programa para que podamos saber cuantas multiplicaciones se han hecho en total entre todos los hilos?

Aquí entra el problema de la sincronización. Supongamos una clase contador muy simple como esta:

```
class Contador{
        int cuenta;
        public Contador(){
                cuenta=0;
        }
        public void incrementar(){
                cuenta=cuenta+1;
        }
        public int getCuenta(){
                return cuenta;
        }
}
```

De esta forma podríamos construir un objeto contador y pasárselo a todos los hilos para que en ese único objeto se almacene el recuento final. El problema es que en la programación multihilo **SI EL OBJETO CONTADOR SE COMPARTE ENTRE VARIOS HILOS LA CUENTA FINAL RESULTANTE ES MUY POSIBLE QUE ESTÉ MAL**

Esta clase debería tener protegidas sus secciones críticas

```
class Contador{
        int cuenta;
        public Contador(){
                cuenta=0;
        }
        public synchronized void incrementar(){
                cuenta=cuenta+1;
        }
        public synchronized int getCuenta(){
                return cuenta;
        }
}
```

Se puede aprovechar todavía más rendimiento si en un método marcamos como sección crítica (o sincronizada) exclusivamente el código peligroso:

```
public  void incrementar(){
        System.out.println("Otras cosas");
        synchronized(this){
                cuenta=cuenta+1;
        }
        System.out.println("Mas cosas...");
        synchronized(this){
                if (cuenta>300){
                        System.out.println("Este hilo trabaja mucho");
                }
        }
}
```

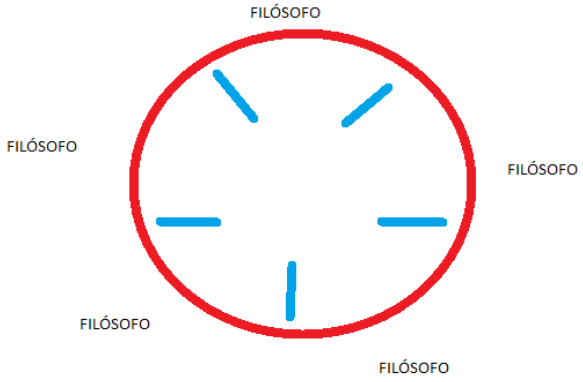
Problema: filósofos

En una mesa hay procesos que simulan el comportamiento de unos filósofos que intentan comer de un plato. Cada filósofo tiene un cubierto a su izquierda y uno a su derecha y para poder comer tiene que conseguir los dos. Si lo consigue, mostrará un mensaje en pantalla que indique «Filosofo 2 comiendo».

Despues de comer, soltará los cubiertos y esperará al azar un tiempo entre 1000 y 5000 milisegundos, indicando por pantalla «El filósofo 2 está pensando».

En general todos los objetos de la clase Filósofo está en un bucle infinito dedicándose a comer y a pensar.

Simular este problema en un programa Java que muestre el progreso de todos sin caer en problemas de sincronización ni de inanición.



Esquema de los filósofos

Boceto de solución

```
import java.util.Random;


public class Filosofo  implements Runnable{
        public void run(){
                String miNombre=Thread.currentThread().getName();
                Random generador=new Random();
                while (true){
                        /* Comer*/
                        /* Intentar coger palillos*/
                        /* Si los coge:*/
                        System.out.println(miNombre+" comiendo...");
                        int milisegs=(1+generador.nextInt(5))*1000;
                        esperarTiempoAzar(miNombre, milisegs);
                        /* Pensando...*/
                        //Recordemos soltar los palillos
                        System.out.println(miNombre+"  pensando...");
                        esperarTiempoAzar(miNombre, milisegs);
                }
        }

private void esperarTiempoAzar(String miNombre, int milisegs) {
                try {
                        Thread.sleep(milisegs);
                } catch (InterruptedException e) {
                        System.out.println(
                                miNombre+" interrumpido!!. Saliendo...");
                        return ;
                }
        }
}
```

Solución completa al problema de los filósofos

Gestor de recursos compartidos (palillos)

```
package com.iesmaestre.filosofos;

public class GestorPalillos {
        boolean palilloLibre[];
        public GestorPalillos(int numPalillos){
                palilloLibre=new  boolean[numPalillos];
```

```
                for (int i=0; i<numPalillos; i++){
                        palilloLibre[i]=true;
                } //Fin del for
        } //Fin del constructor
        public synchronized boolean
                intentarCogerPalillos(int pos1, int pos2)
        {
                boolean seConsigue=false;
                if (
                                (palilloLibre[pos1])
                                &&
                                (palilloLibre[pos2]) )
                {
                        palilloLibre[pos1]=false;
                        palilloLibre[pos2]=false;
                        seConsigue=true;
                } //Fin del if
                return seConsigue;
        }

        public void liberarPalillos(int pos1, int pos2){
                palilloLibre[pos1]=true;
                palilloLibre[pos2]=true;
        }
}
```

Simulación de un filósofo

```
package com.iesmaestre.filosofos;

import java.util.Random;

public class Filosofo implements Runnable{
        GestorPalillos gestorPalillos;
        int posPalilloIzq,posPalilloDer;
        public Filosofo(GestorPalillos g, int pIzq, int pDer){
                this.gestorPalillos=g;
                this.posPalilloDer=pDer;
                this.posPalilloIzq=pIzq;
        }
        public void run() {
                while (true){
                        boolean palillosCogidos;
                        palillosCogidos=
                          this.gestorPalillos.intentarCogerPalillos(
                                        posPalilloIzq, posPalilloDer);
                        if (palillosCogidos){
                                comer();
                                this.gestorPalillos.liberarPalillos(
                                                posPalilloIzq,
                                                posPalilloDer);
                                dormir();
                        } //Fin del if
                } //Fin del while true
        } //Fin del run()

        private void comer() {
                System.out.println("Filosofo "+
                                Thread.currentThread().getName()+
                                " comiendo");
                esperarTiempoAzar();
        }
        private void esperarTiempoAzar() {
                Random generador=new Random();
                int msAzar=generador.nextInt(3);
                try {
                        Thread.sleep(msAzar);
```

```
                } catch (InterruptedException ex) {
                        System.out.println("Fallo la espera");
                }
        }
        private void dormir(){
                System.out.println("Filosofo "+
                                Thread.currentThread().getName()+
                                " durmiendo (zzzzzz)");
                esperarTiempoAzar();
        }
}
```

Lanzador de hilos

```
package com.iesmaestre.filosofos;
public class Lanzador {
        public static void main(String[] args) {
                int NUM_PROCESOS=5;
                Filosofo filosofos[]=new Filosofo[NUM_PROCESOS];
                GestorPalillos gestorPalillos;
                gestorPalillos=new GestorPalillos(NUM_PROCESOS);
                Thread hilos[]=new Thread[NUM_PROCESOS];
                for (int i=1; i<NUM_PROCESOS; i++){
                        filosofos[i]=new Filosofo(
                                gestorPalillos, i, i-1);
                        hilos[i]=new Thread(filosofos[i]);
                        hilos[i].start();
                }
                filosofos[0]=new Filosofo(
                                gestorPalillos, 0, 4);
                hilos[0]=new Thread(filosofos[0]);
                hilos[0].start();
        } //Fin del main
} //Fin de la clase
```

Problema: simulador de casino

Se desea simular los posibles beneficios de diversas estrategias de juego en un casino. La ruleta francesa es un juego en el que hay una ruleta con 37 números (del 0 al 36). Cada 3000 milisegundos el croupier saca un número al azar y los diversos hilos apuestan para ver si ganan. Todos los hilos empiezan con 1.000 euros y la banca (que controla la ruleta) con 50.000. Cuando los jugadores pierden dinero, la banca incrementa su saldo.

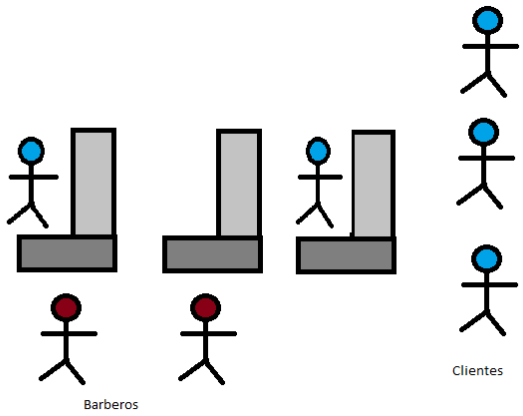
- Se puede jugar a un número concreto. Habrá 4 hilos que eligen números al azar del 1 al 36 (no el 0) y restarán 10 euros de su saldo para apostar a ese ese número. Si sale su número su saldo se incrementa en 360 euros (36 veces lo apostado).
- Se puede jugar a par/impar. Habrá 4 hilos que eligen al azar si apuestan a que saldrá un número par o un número impar. Siempre restan 10 euros para apostar y si ganan incrementan su saldo en 20 euros.
- Se puede jugar a la «martingala». Habrá 4 hilos que eligen números al azar. Elegirán un número y empezarán restando 10 euros de su saldo para apostar a ese número. Si ganan incrementan su saldo en 360 euros. Si pierden jugarán el doble de su apuesta anterior (es decir, 20, luego 40, luego 80, y así sucesivamente)
- La banca acepta todas las apuestas pero nunca paga más dinero del que tiene.
- Si sale el 0, todo el mundo pierde y la banca se queda con todo el dinero.

Problema: barberos

En una peluquería hay barberos y sillas para los clientes (siempre hay más sillas que clientes). Sin embargo, en esta peluquería no siempre hay trabajo por lo que los barberos duermen cuando no hay clientes a los que afeitar. Un cliente puede llegar a la barbería y encontrar alguna silla libre, en cuyo caso, el cliente se sienta y esperará que algún barbero le afeite. Puede ocurrir que el cliente llegue y no haya sillas libres,

en cuyo caso se marcha. Simular el comportamiento de la barbería mediante un programa Java teniendo en cuenta que:

- Se generan clientes continuamente, algunos encuentran silla y otros no. Los que no consigan silla desaparecen (terminan su ejecución)
- Puede haber más sillas que barberos y al revés (poner constantes para poder cambiar fácilmente entre ejecuciones).
- Se recuerda que no debe haber inanición, es decir ningún cliente debería quedarse en una silla esperando un tiempo infinito.



Los barberos dormilones

Una (mala) solución al problema de los barberos

Prueba la siguiente solución:

Clase Barbero

```
public class Barbero implements Runnable {
        private String              nombre;
        private GestorConcurrencia  gc;
        private Random              generador;
        private int                 MAX_ESPERA_SEGS=5;
        public Barbero(GestorConcurrencia gc,String nombre){
                this.nombre     =nombre;
                this.gc         =gc;
                this.generador  =new Random();
        }

        public void esperarTiempoAzar(int max){
                /* Se calculan unos milisegundos al azar*/
                int msgs=(1+generador.nextInt(max))*1000;
                try {
                        Thread.currentThread().sleep(msgs);
                } catch (InterruptedException e) {
                        // TODO Auto-generated catch block
                        e.printStackTrace();
                }
        }
        public void run(){
                while (true){
                        int num_silla=gc.atenderAlgunCliente();
                        while (num_silla==-1){
                                /* Mientras no haya nadie a quien
                                 * atender, dormimos
                                 */
```

```
                                esperarTiempoAzar(MAX_ESPERA_SEGS);
                                num_silla=gc.atenderAlgunCliente();
                        }
                        /* Si llegamos aqui es que había algún cliente
                         * Simulamos un tiempo de afeitado
                         */
                        esperarTiempoAzar(MAX_ESPERA_SEGS);
                        /* Tras ese tiempo de afeitado se
                         * libera la silla
                         */
                        gc.liberarSilla(num_silla);
                        /* Y vuelta a empezar*/
                }
        }
}
```

Clase Cliente

```
public class Cliente implements Runnable{
        GestorConcurrencia      gc;
        public Cliente(GestorConcurrencia gc){
                this.gc         =gc;
        }
        public void run(){
                /* Los clientes no esperan que haya
                 * sillas libres, no hay bucle infinito.
                 * Si no hay sillas libres se van...
                 */
                gc.getSillaLibre();
        }
}
```

Clase GestorConcurrencia

```
public class GestorConcurrencia {
        /* Vector que indica cuantas sillas hay y
         * si están libres o no
         */
        boolean[] sillasLibres;
        /* Indica si el cliente sentado en esa
         * silla está atendido por un barbero o no
         */
        boolean[] clienteEstaAtendido;

        public GestorConcurrencia(int numSillas){
                /*Construimos los vectores...*/
                sillasLibres            =new boolean[numSillas];
                clienteEstaAtendido     =new boolean[numSillas];
                /* ... los inicializamos*/
                for (int i=0; i<numSillas;i++){
                        sillasLibres[i]                 =true;
                        clienteEstaAtendido[i]  =false;
                }
        }

        /**
         * Permite obtener una silla libre, usado por la
         * clase Cliente para saber si puede sentarse
         * en algún sitio o irse
         * @return Devuelve el número de la primera silla
         * que está libre o -1 si no hay ninguna
         */
        public synchronized int getSillaLibre(){
                for (int i=0; i<sillasLibres.length; i++){
                        /* Si está libre la silla ...*/
```

```
                        if (sillasLibres[i]) {
                                /* ...se marca como ocupada*/
                                sillasLibres[i]=false;
                                System.out.println(
                                        "Cliente sentado en silla "+i
                                                );
                                /*.. y devolvemos i...*/
                                return i;
                        }
                }
                /* Si llegamos aquí es que no había nada libre*/
                return -1;
        }

        /**
         * Nos dice qué silla tiene algun cliente
         * que no está atendido
         * @return un número de silla o -1 si no
         * hay clientes sin atender
         */
        public synchronized int atenderAlgunCliente(){
                for (int i=0; i<sillasLibres.length; i++){
                        /* Si una silla está ocupada (no libre, false)
                         * y está marcado como "sin atender" (false)
                         * entonces la marcamos como atendida
                         */
                        if (clienteEstaAtendido[i]==false){
                                clienteEstaAtendido[i]=true;
                                System.out.println(
                                                "Afeitando cliente en silla "+i);
                                return i;
                        }
                }
                return -1;
        }

        /* El cliente de esa silla, se marcha, por lo
         * que se marca esa silla como "libre"
         * y como "sin atender"
         */
        public synchronized void liberarSilla(int i){
                sillasLibres[i]                 =true;
                clienteEstaAtendido[i]  =false;
                System.out.println(
                                "Se marcha el cliente de la silla "+i);
        }
}
```

Clase Lanzador

```
public class Lanzador {

        public static void main(String[] args) {

                int MAX_BARBEROS        =2;
                int MAX_SILLAS          =MAX_BARBEROS+1;
                int MAX_CLIENTES        =MAX_BARBEROS*10;
                int MAX_ESPERA_SEGS     = 3;
                GestorConcurrencia gc;
                gc=new GestorConcurrencia(MAX_SILLAS);

                Thread[] vhBarberos     =new Thread[MAX_BARBEROS];
                for (int i=0; i<MAX_BARBEROS;i++){
                        Barbero b=new Barbero(gc, "Barbero "+i);
                        Thread hilo=new Thread(b);
                        vhBarberos[i]=hilo;
                        hilo.start();
```

```
                }

                /* Generamos unos cuantos clientes
                 * a intervalos aleatorios
                 */
                Random generador=new Random();
                for (int i=0; i<MAX_CLIENTES; i++){
                        Cliente c                       =new Cliente(gc);
                        Thread hiloCliente      =new Thread(c);
                        hiloCliente.start();

                        int msegs=generador.nextInt(3)*1000;
                        try {
                                Thread.sleep(msegs);
                        } catch (InterruptedException e) {
                                // TODO Auto-generated catch block
                                e.printStackTrace();
                        }
                } /* Fin del for*/
        }
}
```

Críticas a la solución anterior

¿Cual es el problema?

El problema está en que la forma que tiene el gestor de concurrencia de decirle a un barbero qué silla tiene un cliente sin afeitar es incorrecta: como siempre se empieza a buscar por el principio del vector, los clientes sentados al final **nunca son atendidos**. Hay que corregir esa asignación para *evitar que los procesos sufran de inanición*.

Clase cliente

```
public class Cliente  {
        GestorSillas gestorSillas;
        public Cliente(GestorSillas g){
                this.gestorSillas = g;
        }
        public void entrarEnBarberia(){
                int posSillaLibre = this.gestorSillas.getPosSillaLibre();
                if (posSillaLibre==-1){
                        System.out.println("No habia sillas libres, me marcho");
                        return ;
                }
                System.out.println("Me siento en la silla:"+posSillaLibre);
        }
}
```

Clase Barbero

```
public class Barbero implements Runnable{
        GestorSillas gestorSillas;
        boolean barberiaAbierta;
        public Barbero (GestorSillas g){
                gestorSillas=g;
                barberiaAbierta=true;
        }

        public void cerrarBarberia(){
                this.barberiaAbierta=false;
        }
        @Override
        public void run() {
           while(barberiaAbierta){
```

```
                int posSillaClienteSinAtender;
                posSillaClienteSinAtender=
                                this.gestorSillas.getSiguienteCliente();
                if (posSillaClienteSinAtender==-1){
                        esperarTiempoAzar(3);
                } else {
                        System.out.println("Barbero atendiendo silla:" +
                                        posSillaClienteSinAtender);
                        esperarTiempoAzar(3);
                        this.gestorSillas.liberarSilla(posSillaClienteSinAtender);
                }
           }
        }

        public static void esperarTiempoAzar(int max){
                Random generador=new Random();
                /* Se calculan unos milisegundos al azar*/
                int msgs=(1+generador.nextInt(max))*1000;
                try {
                                Thread.currentThread().sleep(msgs);
                } catch (InterruptedException e) {
                                // TODO Auto-generated catch block
                                e.printStackTrace();
                }
        }

}
```

Clase GestorSillas

```
public class GestorSillas {
        private int MAX_SILLAS;
        private boolean[] estaSillaLibre;
        private boolean[] clienteEstaAtendido;
        private int siguienteClienteParaAtender=0;
        GestorSillas(int num){
                MAX_SILLAS=num;
                estaSillaLibre=new boolean[MAX_SILLAS];
                clienteEstaAtendido=new boolean[MAX_SILLAS];
                for (int i=0; i<MAX_SILLAS; i++){
                        estaSillaLibre[i]       = true;
                        clienteEstaAtendido[i]  = false;
                }
        }
        /**
         * Nos dice el numero de silla que está libre
         * @return Devuelve una posición o -1 si está
         * todo ocupado
         */
        public synchronized int getPosSillaLibre(){
                int posSilla=-1;
                for (int pos=0; pos<MAX_SILLAS;pos++){
                        if (estaSillaLibre[pos]==true){
                                estaSillaLibre[pos]=false;
                                return pos;
                        }
                }
                return posSilla;
        }

        public void liberarSilla(int pos){
                estaSillaLibre[pos]=true;
                clienteEstaAtendido[pos]=false;
        }
        public synchronized int getSiguienteCliente(){
                int pos=-1;
                boolean salir;
```

```
                int i;
                salir=false;
                i=this.siguienteClienteParaAtender;
                while(!salir){
                        if (
                                        (this.estaSillaLibre[i]==false) &&
                                        (this.clienteEstaAtendido[i]==false)
                        )
                        {
                                this.clienteEstaAtendido[i]=true;
                                this.siguienteClienteParaAtender= (i+1) % MAX_SILLAS;
                                return i;

                        }
                        i++;
                        if (i==this.MAX_SILLAS){
                                i=0;
                        }
                        if (i==this.siguienteClienteParaAtender) salir=true;

                }

                return pos;
        }
}
```

Clase Lanzador

```
public class Lanzador {

        public static void main(String[] args) throws InterruptedException {
                int MAX_BARBEROS = 2;
                int MAX_SILLAS   = 3;
                int MAX_CLIENTES = 1000;
                Barbero[] barberos;
                Thread[]  hilos;

                barberos=new Barbero[MAX_BARBEROS];
                hilos   =new Thread [MAX_BARBEROS];

                GestorSillas gestorSillas=new GestorSillas(MAX_SILLAS);

                for (int i=0; i<MAX_BARBEROS; i++){
                        barberos[i]=new Barbero(gestorSillas);
                        hilos[i]   =new Thread(barberos[i]);

                        hilos[i].start();
                } //Fin del for


                for (int i=0; i< MAX_CLIENTES; i++){
                        Cliente c=new Cliente(gestorSillas);
                        c.entrarEnBarberia();
                }
                Barbero.esperarTiempoAzar(30);
                /* La jornada ha terminado, "cerramos" los barberos*/
                for (int i=0; i<MAX_BARBEROS; i++){
                        barberos[i].cerrarBarberia();
                        hilos[i].join();
                }
                System.out.println("Barberia cerrada.");
        } //Fin del main
} //Fin de la clase
```

Problema: productores y consumidores.

En un cierto programa se tienen procesos que producen números y procesos que leen esos números. Todos los números se introducen en una cola (o vector) limitada.

Todo el mundo lee y escribe de/en esa cola. Cuando un productor quiere poner un número tendrá que comprobar si la cola está llena. Si está llena, espera un tiempo al azar. Si no está llena pone su número en la última posición libre.

Cuando un lector quiere leer, examina si la cola está vacía. Si lo está espera un tiempo al azar, y sino coge el número que haya al principio de la cola y ese número *ya no está disponible para el siguiente*.

Crear un programa que simule el comportamiento de estos procesos evitando problemas de entrelazado e inanición.

Solución

En primer lugar necesitamos una cola que sea «a prueba de hilos» y que permita encolar y desencolar elementos de una manera segura.

Sabemos que antes de encolar hay que comprobar si la cola está llena. También sabemos que antes de desencolar hay que comprobar si la cola está vacía.

Sin embargo **¡¡CUIDADO!!** es posible programar mal aunque las operaciones de nuestra cola estén protegidas con el método «synchronized». Supongamos que un productor tiene código como este:

```
while (true){
        if (!cola.estaLlena()){
                cola.encolar(4);
        }
}
```

Puede que pensemos que este código está bien pero **ESTÁ MAL**. Está mal porque puede que en el tiempo transcurrido entre que un productor ejecuta `estaLlena` y `encolar` OTRO HILO SE HAYA COLADO ENTRE MEDIAS Y HAYA ALTERADO LA EJECUCIÓN. Eso significa que **aunque llamemos a dos métodos synchronized uno detrás de otro es posible que la ejecución sea incorrecta**. Así, necesitaremos hacer operaciones `encolar` y `desencolar` que funcionen de manera atómica y nos avisen de si consiguen hacer la operación o no.

Una cola limitada en tamaño y thread-safe

```
public class Cola {
        private int MAX_ELEMENTOS;
        LinkedList<Integer> cola;
        public Cola (int max){
                cola=new LinkedList<Integer>();
                this.MAX_ELEMENTOS=max;
        }
        /* En realidad, si estamos seguro de que nadie
        llamará a este método podriamos ponerla como no
        synchronized*/
        public synchronized boolean estaVacia(){
                int numElementos=cola.size();
                if (numElementos==0){
                        return true;
                }
                return false;
        }
        /* Igual que antes, si estamos seguro de que nadie
        llamará a este método podriamos ponerla como no
        synchronized*/
        public synchronized boolean estaLlena(){
                int numElementos=cola.size();
                if (numElementos==this.MAX_ELEMENTOS){
```

```
                        return true;
                }
                return false;
        }
        /* Devuelve true si se pudo hacer y false si no se pudo*/
        public synchronized boolean encolar(int numero){
                if (estaLlena()){
                        return false;
                }
                cola.addLast(numero);
                return true;
        }
        public synchronized int desencolar(){
                /* Necesitamos un número especial que actúe
                como comprobador de errores*/
                if (estaVacia()){
                        return -1;
                }
                int numero=cola.removeFirst();
                return numero;
        }

}
```

La clase Productor

```
public class Productor implements Runnable{
        Cola colaCompartida;
        public Productor(Cola cola){
                this.colaCompartida=cola;
        }
        public void run() {
           while (true){
                   int num=Utilidades.numAzar(10);
                   while (colaCompartida.encolar(num)==false){
                           Utilidades.esperarTiempoAzar(3000);
                   } /*Fin del while*/

                   System.out.println("Productor encoló el numero:"+num);
           } /*Fin del while externo*/
        } /*Fin de run()*/
} /*Fin de la clase*/
```

La clase Consumidor

```
public class Consumidor implements Runnable{
        Cola colaCompartida;
        public Consumidor(Cola cola){
                this.colaCompartida=cola;
        }
        @Override
        public void run() {
                int num;
                while (true){
                        num=colaCompartida.desencolar();
                        if (num!=-1){
                                System.out.println("Consumidor recuperó el numero:"+num);
                        } /* Fin del if*/
                } /*Fin del bucle infinito*/
        } /* Fin del run()*/
} /*Fin de la clase Consumidor*/
```

Un lanzador

```
public class Lanzador {

	public static void main(String[] args) throws InterruptedException {
		int MAX_PRODUCTORES     = 5;
		int MAX_CONSUMIDORES    = 7;
		int MAX_ELEMENTOS       = 10;

		Thread[] hilosProductor;
		Thread[] hilosConsumidor;

		hilosProductor   = new Thread[MAX_PRODUCTORES];
		hilosConsumidor  = new Thread[MAX_CONSUMIDORES];

		Cola colaCompartida=new Cola(MAX_ELEMENTOS);

		/*Construimos los productores*/
		for (int i=0; i<MAX_PRODUCTORES; i++){
			Productor productor=new Productor(colaCompartida);
			hilosProductor[i]=new Thread(productor);
			hilosProductor[i].start();
		}
		/*Construimos los consumidores*/
		for (int i=0; i<MAX_CONSUMIDORES; i++){
			Consumidor consumidor=new Consumidor(colaCompartida);
			hilosConsumidor[i]=new Thread(consumidor);
			hilosConsumidor[i].start();
		}

		/* Esperamos a que acaben todos los hilos, primero
		productores y luego consumidores
		*/
		for (int i=0; i<MAX_PRODUCTORES; i++){
			hilosProductor[i].join();
		}
		for (int i=0; i<MAX_CONSUMIDORES; i++){
			hilosConsumidor[i].join();
		}
	}

}
```

Ejercicio

En unos grandes almacenes hay 300 clientes agolpados en la puerta para intentar conseguir un producto del cual solo hay 100 unidades.

Por la puerta solo cabe una persona, pero la paciencia de los clientes es limitada por lo que solo se harán un máximo de 10 intentos para entrar por la puerta. Si despues de 10 intentos la puerta no se ha encontrado libre ni una sola vez, el cliente desiste y se marcha.

Cuando se consigue entrar por la puerta el cliente puede encontrarse con dos situaciones:

1. Quedan productos: el cliente cogerá uno y se marchará.
2. No quedan productos: el cliente simplemente se marchará.

Realizar la simulación en Java de dicha situación.

Una posible solución sería la siguiente:

Clase cliente

```
public class Cliente implements Runnable{

        Puerta      puerta;
        Almacen     almacen;
        String      nombre;
        Random      generador;
        final int MAX_INTENTOS  =   10;
        public Cliente(Puerta p, Almacen a, String nombre){
                this.puerta     =   p;
                this.almacen    =   a;
                this.nombre     =   nombre;
                generador       =   new Random();
        }

        public void esperar(){
                try {
                        int ms_azar = generador.nextInt(100);
                        Thread.sleep(ms_azar);
                } catch (InterruptedException ex) {
                        System.out.println("Falló la espera");
                }
        }
        @Override
        public void run() {
                for (int i=0; i<MAX_INTENTOS; i++){
                        if (!puerta.estaOcupada()){
                                if (puerta.intentarEntrar()){
                                        esperar();
                                        puerta.liberarPuerta();
                                        if (almacen.cogerProducto()){
                                                System.out.println(
                                                                this.nombre + ": cogí un prc
                                                return ;
                                        }
                                        else {
                                                System.out.println(
                                                          this.nombre+
                                                          ": ops, crucé pero no cogí nada")
                                                return ;
                                        } //Fin del else
                                } //Fin del if
                        } else{
                           esperar();
                        }

                } //Fin del for
                //Se superó el máximo de reintentos y abandonamos
                System.out.println(this.nombre+
                                ": lo intenté muchas veces y no pude");
        }

}
```

Clase Almacén

```
package grandesalmacenes;

public class Almacen {
        private int numProductos;
        public Almacen(int nProductos){
                this.numProductos=nProductos;
        }
        public boolean cogerProducto(){
                if (this.numProductos>0){
                        this.numProductos--;
```

```
                        return true;
                }
                return false;
        }
}
```

Clase Puerta

```
package grandesalmacenes;

import java.util.Random;
import java.util.logging.Level;
import java.util.logging.Logger;


public class Puerta {
        boolean ocupada;

        Puerta(){
                this.ocupada=false;

        }
        public boolean estaOcupada(){
                return this.ocupada;
        }
        public synchronized void liberarPuerta(){
                this.ocupada=false;
        }
        public synchronized boolean intentarEntrar(){
                if (this.ocupada) return false;
                /* Si llegamos aquí, la puerta estaba libre
                pero la pondremos a ocupada un tiempo
                y luego la volveremos a poner a "libre"*/
                this.ocupada=true;
                return true;
        }
}
```

Clase lanzadora (GrandesAlmacenes)

```
        }
    }
```

Simulación bancaria

Un banco necesita controlar el acceso a cuentas bancarias y para ello desea hacer un programa de prueba en Java que permita lanzar procesos que ingresen y retiren dinero a la vez y comprobar así si el resultado final es el esperado.

Se parte de una cuenta con 100 euros y se pueden tener procesos que ingresen 100 euros, 50 o 20. También se pueden tener procesos que retiran 100, 50 o 20 euros euros. Se desean tener los siguientes procesos:

- 40 procesos que ingresan 100
- 20 procesos que ingresan 50
- 60 que ingresen 20.

De la misma manera se desean lo siguientes procesos que retiran cantidades.

- 40 procesos que retiran 100
- 20 procesos que retiran 50
- 60 que retiran 20.

Se desea comprobar que tras la ejecución la cuenta tiene exactamente 100 euros, que era la cantidad de la que se disponía al principio. Realizar el programa Java que demuestra dicho hecho.

En primer lugar la clase Cuenta podría ser así:

```
public class Cuenta {

    private int saldo;
    private int saldoInicial;

    public Cuenta(int saldo){
        this.saldoInicial=saldo;
        this.saldo=saldo;
    }
    public synchronized void hacerMovimiento(int cantidad){
        this.saldo = this.saldo+cantidad;
    }
    public boolean esSimulacionCorrecta(){
        if (this.saldo==this.saldoInicial) return true;
        return false;
    }
    public int getSaldo(){
        return this.saldo;
    }
}
```

La clase HiloCliente podría ser así:

```
public class HiloCliente implements Runnable{
    Cuenta cuenta;
    int cantidad;

    public HiloCliente(Cuenta cuenta, int cantidad) {
        this.cuenta = cuenta;
        this.cantidad = cantidad;
    }

    @Override
    public void run() {
        /* Forzamos la maquinaria: repetimos
```

```
            la operación muchísimas veces para
            intentar verificar si la simulación es
            correcta
            */
            for (int i=0; i<100; i++){
                cuenta.hacerMovimiento(cantidad);
            }
        }
    }
```

La clase Lanzador podría ser así:

```
    public class Lanzador {
        public static void main(String[] args) throws InterruptedException{
            Cuenta cuenta = new Cuenta (100);

            final int NUM_OPS_CON_100 = 40;
            final int NUM_OPS_CON_50  = 20;
            final int NUM_OPS_CON_20  = 60;

            Thread[] hilosIngresan100 = new Thread[NUM_OPS_CON_100];
            Thread[] hilosRetiran100  = new Thread[NUM_OPS_CON_100];
            Thread[] hilosIngresan50  = new Thread[NUM_OPS_CON_50];
            Thread[] hilosRetiran50   = new Thread[NUM_OPS_CON_50];
            Thread[] hilosIngresan20  = new Thread[NUM_OPS_CON_20];
            Thread[] hilosRetiran20   = new Thread[NUM_OPS_CON_20];

            /* Arrancamos todos los hilos*/
            for (int i=0; i<NUM_OPS_CON_100;i++){
                HiloCliente ingresa = new HiloCliente(cuenta, 100);
                HiloCliente retira  = new HiloCliente(cuenta, -100);

                hilosIngresan100[i]= new Thread(ingresa);
                hilosRetiran100[i] = new Thread(retira);

                hilosIngresan100[i].start();
                hilosRetiran100[i].start();
            }

            for (int i=0; i<NUM_OPS_CON_50;i++){
                HiloCliente ingresa = new HiloCliente(cuenta, 50);
                HiloCliente retira  = new HiloCliente(cuenta, -50);

                hilosIngresan50[i]= new Thread(ingresa);
                hilosRetiran50[i] = new Thread(retira);

                hilosIngresan50[i].start();
                hilosRetiran50[i].start();
            }

            for (int i=0; i<NUM_OPS_CON_20;i++){
                HiloCliente ingresa = new HiloCliente(cuenta, 20);
                HiloCliente retira  = new HiloCliente(cuenta, -20);

                hilosIngresan20[i]= new Thread(ingresa);
                hilosRetiran20[i] = new Thread(retira);

                hilosIngresan20[i].start();
                hilosRetiran20[i].start();
            }

            /* En este punto todos los hilos están arrancados,
            ahora toca esperarlos */

            for (int i=0; i<NUM_OPS_CON_100;i++){
                hilosIngresan100[i].join();
                hilosRetiran100[i].join();
            }
```

```
        for (int i=0; i<NUM_OPS_CON_50;i++){
            hilosIngresan50[i].join();
            hilosRetiran50[i].join();
        }

        for (int i=0; i<NUM_OPS_CON_20;i++){
            hilosIngresan20[i].join();
            hilosRetiran20[i].join();
        }
        if (cuenta.esSimulacionCorrecta()){
            System.out.println("La simulación fue correcta");
        } else {
            System.out.println("La simulación falló ");
            System.out.println("La cuenta tiene:"+
                    cuenta.getSaldo());
            System.out.println("Revise sus synchronized");
        }

    }
}
```

Clases de alto nivel: la clase `RecursiveTask`

En Java 8 y posteriores se puede utilizar una clase llamada `RecursiveTask` que facilita un poco la tarea de crear aplicaciones con paralelismo. La idea básica es que podemos heredar de esta clase indicando qué tipo Java tendrá el resultado de la operación. Dentro de la clase deberemos implementar un método `compute` que básicamente deberá trabajar recursivamente decidiendo si la tarea es lo bastante pequeña para ejecutarla o si la dividimos más.

Supongamos que queremos sumar todos los números de un vector. Por comodidad supongamos también que siempre nos pasan vectores con un tamaño que es potencia de 2. Podemos usar una clase Java como esta para hacer la operación de forma paralela.

```
package io.github.oscarmaestre.paralelismo;

import java.util.concurrent.RecursiveTask;

public class Sumador extends RecursiveTask<Long>{
    int[] numeros;
    int pos1, pos2;
    final int NUM_MAX_ELEMENTOS=2;
    private int longitud;
    private int posMitad;
    private int limiteIzquierdo;
    private int limiteDerecho;
    /*Por simplificar solo aceptamos vectores
    con un número de elementos que sea una
    potencia de 2*/
    public Sumador(int[] numeros, int pos1, int pos2)  {
        this.numeros    = numeros;
        this.pos1       = pos1;
        this.pos2       = pos2;
        longitud        = 1 + (pos2- pos1);
        posMitad        = longitud/2;
        limiteIzquierdo = pos1+(posMitad-1);
        limiteDerecho   = limiteIzquierdo+1;
    }

    @Override
    protected Long compute() {
        /* Si el vector tiene estos elementos,
        simplemente sumamos*/
        if (longitud==NUM_MAX_ELEMENTOS){
            long suma=0;
```

```
            for (int i=pos1; i<=pos2; i++){
                suma=suma + numeros[i];
            }
            return suma;
        }

        /*Pero si tiene más elementos,
        dividimos el vector en dos y sumamos
        las dos mitades de forma independiente
        y paralela*/
        Sumador sumadorIzq=
                new Sumador(numeros, pos1, limiteIzquierdo);
        Sumador sumadorDer=
                new Sumador(numeros, limiteDerecho, pos2);

        /*Lanzamos los sumadores...*/
        sumadorIzq.fork();
        sumadorDer.fork();

        /* Y recogemos sus resultados*/
        long sumaIzq=sumadorIzq.join();
        long sumaDer=sumadorDer.join();

        return sumaIzq + sumaDer;
    } /*Fin del compute*/
} /*Fin de la clase*/
```

Con esta clase como «lanzador de hilos», nuestro trabajo se simplifica bastante. Ahora podemos tener un `main` que va a ser tan sencillo como el código siguiente:

```
package io.github.oscarmaestre.paralelismo;

import java.util.concurrent.ExecutionException;
import java.util.concurrent.ForkJoinPool;

public class SumadorParalelo {

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

        /* Construimos un vector de ejemplo...*/
        final int MAX_NUMEROS=1024;
        int[] numeros=new int[MAX_NUMEROS];

        /*Y lo rellenamos con números*/
        for (int i=0; i<MAX_NUMEROS; i++){
            numeros[i]=i;
        }

        /* Esta clase gestionará el paralelismo de las tareas*/
        ForkJoinPool pool=new ForkJoinPool();

        /* Fabricamos un sumador inicial que intente sumar todo*/
        Sumador sumador=new Sumador(numeros, 0, MAX_NUMEROS-1);

        /*Y la clase ForkJoinPool invocará a nuestro sumador lanzando
        los hilos, recogiendo los resultados y haciendo todo lo necesario
        para que al final solo tengamos que recoger el resultado*/
        pool.invoke(sumador);

        /* Resultado que podemos ver aquí*/
        Long resultado = sumador.get();
        System.out.println("La suma es:"+resultado);
    } /*Fin del main*/
} /*Fin de la clase*/
```

Documentación.

La estructura general de toda aplicación multihilo es algo similar a lo siguiente:

- Habrá tareas que se puedan paralelizar. Dichas tareas heredarán de `Runnable`.
- Habrá procesos que pueden ser accedidos por muchos hilos. Estos objetos **deberán usar synchronized**
- Habrá que crear los recursos compartidos y crear los hilos. Esta operación se debe hacer en este orden.

Depuración.

La depuración de programas multihilo debe hacerse exclusivamente trabajando con pruebas de unidad y las comprobaciones de tipos de Java.

- Por un lado, los depuradores solo funcionan vinculándose con el hilo principal («main») de la aplicación, por lo que si se desea depurar un hilo concreto se tendrá que averiguar el identificador de dicho hilo para poder conectar con él desde el depurador.
- Los depuradores «interfieren» con la ejecución de los hilos así que pueden pasar cosas tan curiosas como que un programa *funcione cuando usamos el depurador pero vuelva a fallar al ejecutarlo solo*.
- Los mensajes de traza pueden aparecer antes o despues de lo esperado, así que en realidad no sabemos cuando se escribió un mensaje en pantalla ni qué operaciones se efectuaron antes o despues.

Ejercicio: empleado «Thread safe»

Se dice que un método o clase es «Thread safe» (a prueba de hilos) cuando se ha programado con cuidado para que cualquier persona que lo utilice pueda hacerlo desde múltiples hilos de ejecución sin miedo a que se produzcan resultados indeseados. Evidentemente, la programación de código «Thread-safe» involucra muchísimo más cuidado de lo habitual. En el enunciado siguiente se pide desarrollar una clase que claramente debe ser «Thread safe».

Crear una clase multihilo llamada Empleado que permita cambiar la cantidad de horas trabajadas por parte del empleado y su sueldo permitiendo que haya muchos hilos a la vez que puedan cambiar ambos valores pero de manera que la ejecución dé siempre resultados coherentes.

En concreto, se desea tener un método `incrementarHoras(int numHoras)` que acepte números positivos y negativos y que permita actualizar el número de horas trabajadas. También se desea un método `incrementarBonus (int bonus)` que acepte positivos y negativos y que permita incrementar el bonus salarial del trabajador.

Si por ejemplo lanzamos 10 hilos con 20 ejecuciones cada uno y todos ellos llaman a `incrementarHoras(2)` la ejecución de estos hilos debería reflejar que el empleado ha trabajado 400 horas este mes. De la misma manera, si hay 10 hilos que ejecuta cada uno 10 veces el método `incrementarBonus(10)` el bonus salarial del empleado debería ser de 1000 euros.

```
public class Empleado {
        int bonus=0;
        int horasTrabajadas=0;
        public synchronized void incrementarHoras(int numHoras){
                horasTrabajadas+=numHoras;
        }
        public synchronized void incrementarBonus(int bonusSalarial){
                bonus+=bonusSalarial;
        }
}
```