



**Dept. de Ingeniería de Sistemas Telemáticos
E.T.S.I. de Telecomunicación
Universidad Politécnica
Madrid**

Java

Vademécum /concurrency

José A. Mañas
12 de abril de 2012

Índice

INTRODUCCIÓN	5
VADEMÉCUM	6
1. CALL (MÉTODO) V CALL()	6
1.1. <i>Ejemplo</i>	6
2. CARRERAS [RACE CONDITIONS] (CONCEPTO)	8
3. CERROJO [LOCK] (CONCEPTO).....	8
3.1. <i>Ejemplo</i>	8
4. COLAS DE COMUNICACIÓN ENTRE THREADS.....	9
5. CONCURRENCIA (CONCEPTO).....	9
6. CONCURRENT COLLECTIONS (CLASES).....	10
7. CONDICIÓN [CONDITION] (CONCEPTO).....	10
7.1. <i>Ejemplo</i>	10
8. DEADLOCK [INTERBLOQUEO] (CONCEPTO)	13
9. DEMONIO [DAEMON] (CONCEPTO).....	13
10. EJECUTOR [EXECUTOR] (CONCEPTO).....	13
10.1. <i>Ejemplo</i>	14
11. EQUIDAD [FAIRNESS] (CONCEPTO)	15
12. EXCEPCIONES [EXCEPTIONS]	15
13. FAIRNESS [EQUIDAD] (CONCEPTO)	16
14. HEBRA [THREAD] (CONCEPTO).....	16
15. HILO [THREAD] (CONCEPTO)	16
16. ILLEGALMONITORSTATEEXCEPTION [EXCEPCIÓN] JAVA.LANG.....	16
17. INANICIÓN [STARVATION] (CONCEPTO)	17
18. INTERBLOQUEO [DEADLOCK] (CONCEPTO)	17
19. INTERRUPCIONES (CONCEPTO).....	17
19.1. <i>Ejemplo</i>	18
20. INTERRUPT (MÉTODO) VOID INTERRUPT()	19
21. INTERRUPTED (MÉTODO) STATIC BOOLEAN INTERRUPTED().....	19
22. INTERRUPTEDEXCEPTION (EXCEPCIÓN).....	19
23. ISINTERRUPTED (MÉTODO) BOOLEAN ISINTERRUPTED().....	20
24. JOIN (MÉTODO) VOID JOIN().....	20
25. LIVELOCK (CONCEPTO)	21
26. LIVENESS [VIVACIDAD] (CONCEPTO).....	21
27. LOCK [CERROJO] (CONCEPTO).....	21
28. MONITOR (CONCEPTO)	21
29. MUTEX (CONCEPTO).....	21
30. NOTIFY (MÉTODO) VOID NOTIFY()	22
31. NOTIFYALL (MÉTODO) VOID NOTIFYALL()	22
32. OPERACIÓN ATÓMICA (CONCEPTO)	22
33. RACE CONDITIONS [CARRERAS] (CONCEPTO)	22
34. REENTRANTE [REentrant] (CONCEPTO)	23
35. RUN (MÉTODO) VOID RUN()	23
36. SAFETY [SEGURIDAD] (CONCEPTO)	23
37. SECCIÓN CRÍTICA	23
38. SEGURIDAD [SAFETY] (CONCEPTO)	23
39. SEMÁFORO [SEMAPHORE] (CONCEPTO)	24
39.1. <i>Ejemplo: semáforo binario</i>	24
39.2. <i>Ejemplo: semáforo con N permisos</i>	25
40. SINCRONIZADORES (CONCEPTO)	26

41.	SLEEP (MÉTODO) VOID SLEEP(LONG)	26
42.	START (MÉTODO) VOID START()	27
43.	STARVATION [INANICIÓN] (CONCEPTO)	27
44.	STOP (DEPRECATED) VOID STOP()	27
45.	SYNCHRONIZED (PALABRA RESERVADA)	27
	Zonas de exclusión mutua	27
	Métodos sincronizados	27
	Terminación abrupta	28
	Variables volatile en zonas synchronized	28
45.2.	<i>Ejemplo: bloque de sentencias sincronizado</i>	28
45.3.	<i>Ejemplo: cuenta corriente</i>	29
	Solución con zonas de exclusión mutua	30
	Solución con métodos sincronizados	30
46.	THREAD [HILO] (CONCEPTO)	31
	¿Cómo se crea un thread? Opción 1: new Thread(Runnable)	31
	¿Cómo se crea un thread? Opción 2: extends Thread	32
	¿Cómo se arranca un thread?	33
	¿Cuándo termina un thread?	33
	¿Cuándo acaba el programa?	33
	¿Cómo se para un thread?	33
	¿Cómo espera un thread a que otro termine?	35
	¿Cómo se le pasan valores a un thread?	35
47.	THREAD-SAFE (CONCEPTO)	35
48.	VARIABLES ATÓMICAS	35
49.	VIVACIDAD [LIVENESS] (CONCEPTO)	36
50.	VOLATILE (PALABRA RESERVADA)	36
	volatile en synchronized	36
51.	WAIT() (MÉTODO)	36
51.1.	<i>while(condición) { wait(); }</i>	36
52.	YIELD() (MÉTODO) VOID YIELD()	37
52.1.	<i>Ejemplo</i>	37
53.	ZONA CRÍTICA (CONCEPTO)	38
54.	ZONAS DE EXCLUSIÓN MUTUA (CONCEPTO)	38
54.1.	<i>Ejemplo</i>	39
LAS BIBLIOTECAS DE JAVA		40
1.	ATOMICBOOLEAN (CLASS) JAVA.UTIL.CONCURRENT.ATOMIC	40
2.	ATOMICINTEGER (CLASS) JAVA.UTIL.CONCURRENT.ATOMIC	40
3.	ATOMICINTEGERARRAY (CLASS) JAVA.UTIL.CONCURRENT.ATOMIC	41
4.	ATOMICLONG (CLASS) JAVA.UTIL.CONCURRENT.ATOMIC	42
5.	ATOMICLONGARRAY (CLASS) JAVA.UTIL.CONCURRENT.ATOMIC	42
6.	BLOCKINGQUEUE<E> (INTERFACE) JAVA.UTIL.CONCURRENT	42
6.1.	<i>Ejemplo</i>	42
7.	CALLABLE<V> (INTERFACE) JAVA.UTIL.CONCURRENT	44
8.	CONCURRENTMAP<K, V> (INTERFACE) JAVA.UTIL.CONCURRENT	44
9.	CONDITION (INTERFACE) JAVA.UTIL.CONCURRENT.LOCKS	44
10.	COPYONWRITEARRAYLIST<E> (CLASS) JAVA.UTIL.CONCURRENT	44
10.1.	<i>Ejemplo</i>	45
11.	COPYONWRITEARRAYSET<E> (CLASS) JAVA.UTIL.CONCURRENT	46
12.	COUNTDOWNLATCH (CLASS) JAVA.UTIL.CONCURRENT	46
12.1.	<i>Ejemplo 1. Señal de salida: arranque sincronizado</i>	46
12.2.	<i>Ejemplo 2. Señal de fin: sincronización de llegada</i>	47
13.	CYCLICBARRIER (CLASS) JAVA.UTIL.CONCURRENT	48
13.1.	<i>Ejemplo</i>	48
14.	EXCHANGER (CLASS) JAVA.UTIL.CONCURRENT	49

14.1. Ejemplo.....	49
15. EXECUTOR (INTERFACE) JAVA.UTIL.CONCURRENT	50
16. EXECUTORS (FACTORY) JAVA.UTIL.CONCURRENT.....	50
17. EXECUTORSERVICE (INTERFACE) JAVA.UTIL.CONCURRENT.....	51
18. FUTURE<V> (INTERFACE)	51
19. LOCK (INTERFACE) JAVA.UTIL.CONCURRENT.LOCKS.....	52
20. PIPEDINPUTSTREAM (CLASS) JAVA.IO	53
21. PIPEDOUTPUTSTREAM (CLASS) JAVA.IO	53
22. PIPEDREADER (CLASS) JAVA.IO	53
23. PIPEDWRITER (CLASS) JAVA.IO	54
24. READWRITELOCK (INTERFACE) JAVA.UTIL.CONCURRENT.LOCKS.....	54
25. RUNNABLE (INTERFACE) JAVA.LANG	54
26. SCHEDULEDEXECUTORSERVICE (INTERFACE) JAVA.UTIL.CONCURRENT	54
27. SCHEDULEDFUTURE<V> (INTERFACE) JAVA.UTIL.CONCURRENT	55
28. SEMAPHORE (CLASS) JAVA.UTIL.CONCURRENT.....	55
29. THREAD (CLASE) JAVA.LANG	55
Estado	56
Prioridad.....	56
30. THREADGROUP (CLASS) JAVA.LANG.....	57
31. TIMER (CLASS) JAVA.UTIL.....	57
31.1. Ejemplo.....	57
32. TIMERTASK (CLASS) JAVA.UTIL.....	58
PROBLEMAS CLASICOS.....	59
1. DINING PHILOSOPHERS (PROBLEMA).....	59
Solución 1: monitor.....	59
Solución 2: semáforo.....	61
2. PRODUCTORES Y CONSUMIDORES (PROBLEMA)	63
Solución 1: wait & notify	64
Solución 2: con semáforos	65
Solución 3: locks & conditions.....	66
Solución 4: BlockingQueue.....	67
Solución 5: Piped streams	68
3. READERS & WRITERS (PROBLEMA).....	68
Solución: monitor con condiciones	69
Solución: ReadWriteLock	71
4. SINGLETON (PATTERN)	72
DICCIONARIO	74
1. ACRÓNIMOS	74
2. TÉRMINOS EN INGLÉS	74

Introducción

(Del latín *vade*, anda, ven, y *mecum*, conmigo).

1. m. Libro de poco volumen y de fácil manejo para consulta inmediata de nociones o informaciones fundamentales.
2. m. Cartapacio en que los niños llevaban sus libros y papeles a la escuela.

Java es un lenguaje de programación orientado a objetos, como otros muchos, con los que comparte los conceptos fundamentales. Pero, además de los conceptos, cada lenguaje de programación conlleva multitud de pequeños detalles que a menudo se olvidan.

Estas notas repasan los detalles conceptuales y sintácticos de los programas concurrentes escrito en Java. Se han ordenado alfabéticamente para que sean fáciles de localizar. Cada entrada se explica someramente y se incluyen pequeños ejemplos ilustrativos de cómo se usa.

Algunas entradas se refieren a clases de la biblioteca java; en estos casos se comentan los elementos más frecuentemente utilizados. Para mayor detalle, hay que recurrir a la especificación completa.

Derechos de autor

© 2012, José A. Mañas <jmanas@dit.upm.es>

El autor cede a los lectores el derecho de reproducir total o parcialmente cualquier parte de este documento siempre y cuando se cite la fuente y la reproducción sea fiel al original o se indique que ha sido modificado con las palabras "*inspirado en*".

El autor cede a los lectores el derecho de utilizar el código incluido en los ejemplos, incluso comercialmente, siempre y cuando se cite la fuente. No obstante, el autor declina responsabilidad alguna respecto de la idoneidad del código para fin alguno o la posible existencia de errores que pudieran repercutir en sus usuarios. Cualquier uso del código que aparece en este documento será bajo la entera responsabilidad del usuario del mismo.

Vademécum

Las entradas están por orden alfabético, y repartidas entre esta sección, más conceptual, y la parte siguiente que describe clases de la biblioteca de java.

No está pensado para leerlo empezando por la página 1, sino más bien para ir saltando de enlace en enlace.

Quizás los ejemplos se pueden ordenar desde lo más sencillo hacia conceptos más avanzados:

- creación y lanzamiento de threads
 1. [extends Thread](#)
 2. [new Thread\(Runnable\)](#)
 3. [ejecutor](#)Runnable + Executor.execute()
 4. [Callable](#) Callable<V> + Executor.execute() + Future<V>
- el caso de los [productores-consumidores](#)
- [zonas críticas o de exclusión mutua](#)
 1. [zonas synchronized](#)
 2. [métodos synchronized](#)
 3. [variables atómicas](#)
 4. usando [Lock](#)
- el caso de los filósofos: [dining philosophers](#)
- el caso de los lectores y escritores: [readers & writers](#)

1. **call (método) V call()**

Método de la interfaz [Callable<V>](#).

Hay un paralelismo entre

<code>interface Runnable</code>	<code>interface Callable<V></code>
<code>void run()</code>	<code>V call() throws Exception</code>

Ver "[ExecutorService](#)".

Ver "[Future](#)".

1.1. Ejemplo

Se trata de sumar todos los términos de una matriz de enteros.

Para ello lanzaremos un thread para sumar cada fila:

class Filtro
<pre>import java.util.concurrent.Callable; public class Filtro implements Callable<Integer> { private int[] fila; public Filtro(int[] fila) { this.fila = fila; } }</pre>

```

@Override
public Integer call() throws Exception {
    int total = 0;
    for (int n : fila)
        total += n;
    return total;
}
}

```

Ahora organizamos un programa con varios threads. A cada uno le asignamos la tarea de sumar una fila. Al acabar, recopilamos los resultados y agregamos la suma total:

class Matriz

```

import java.util.Arrays;
import java.util.concurrent.*;

public class Matriz {
    private static final int FILAS = 40;
    private static final int COLUMNAS = 4;

    private static final int POOL_SIZE = 3;

    @SuppressWarnings("unchecked")
    public static void main(String[] args)
        throws InterruptedException, ExecutionException {
        // preparamos una matriz aleatoria
        int[][] matriz = new int[FILAS][COLUMNAS];
        for (int fila = 0; fila < FILAS; fila++) {
            for (int col = 0; col < COLUMNAS; col++) {
                matriz[fila][col] =
                    (int) (Math.random() * 1000);
            }
        }

        // para ir recopilando resultados de los threads
        Future<Integer>[] resultados = new Future[FILAS];

        // conjunto de ejecutores
        ExecutorService pool =
            Executors.newFixedThreadPool(POOL_SIZE);

        // lanzamos todas los threads
        for (int fila = 0; fila < FILAS; fila++) {
            Callable<Integer> filtro =
                new Filtro(matriz[fila]);
            Future<Integer> future = pool.submit(filtro);
            resultados[fila] = future;
        }

        // recopilamos resultados y agregamos
        int suma = 0;
        for (int fila = 0; fila < FILAS; fila++) {
            Future<Integer> future = resultados[fila];
            Integer parcial = future.get();
            suma += parcial;
        }
        System.out.println("suma total: " + suma);
        pool.shutdown();
    }
}

```

```
}  
}
```

2. Carreras [race conditions] (concepto)

Ver "[race conditions](#)".

Ver "[seguridad](#)".

3. Cerrojo [lock] (concepto)

Es probablemente el mecanismo más básico para controlar qué cosas se pueden ejecutar en cualquier orden y qué cosas requieren una cierta disciplina.

Los cerrojos delimitan una zona de código (ver [zonas de exclusión mutua](#))

```
cerrojo.lock();  
// zona protegida  
cerrojo.unlock();
```

El cerrojo es propiedad de un solo thread en cada momento. Si un thread quiere hacerse con el cerrojo y lo tiene otro, espera a que lo libere.

Los cerrojos son mecanismos de sincronización de bajo nivel pues es fácil escribir programas erróneos con ellos.

Los cerrojos dependen casi siempre de la buena voluntad de los threads que los usan: si un thread no se ajusta a la disciplina de adquirir el cerrojo antes de meterse en una zona crítica, pues los demás threads no pueden hacer nada por evitarlo. Es por ello que se recomienda fuertemente que las zonas críticas se encapsulen de forma que sea imposible acceder a su contenido sin control.

Igualmente hay que contar con el buen hacer de los threads que usan los cerrojos. Si un thread se olvida de liberarlo, queda cerrado. Por ejemplo, si salta una interrupción y el código de tratamiento no abre el cerrojo.

Ver "[Lock](#)".

Ver "[zonas de exclusión mutua](#)".

3.1. Ejemplo

```
class ContadorCompartido  
  
public class ContadorCompartido {  
    private int n = 0;  
  
    public int getN(String id) {  
        return n;  
    }  
  
    public void setN(String id, int n) {  
        this.n = n;  
        System.err.println(id + ": " + n);  
    }  
}  
  
class IncrementadorLento  
  
import java.util.concurrent.locks.Lock;  
import java.util.concurrent.locks.ReentrantLock;
```

```

public class IncrementadorLento extends Thread {
    private final String id;
    private final ContadorCompartido cc;

    private static Lock lock = new ReentrantLock();

    public IncrementadorLento(String id, ContadorCompartido cc) {
        this.id = id;
        this.cc = cc;
    }

    @Override
    public void run() {
        llave.lock();
        try {
            int valor = cc.getN(id);
            valor++;
            sleep(1000);
            cc.setN(id, valor);
        } catch (InterruptedException e) {
            System.err.println(id + ": " + e);
        } finally {
            llave.unlock();
        }
    }
}

```

4. Colas de comunicación entre threads

Java ofrece varias opciones para que los threads se comuniquen entre sí estableciendo una cola entre ellos. La idea general es que dicha cola

- tiene capacidad para almacenar uno o más datos
- bloquea los threads que quieren añadir datos si no hay sitio para retenerlos
- bloquea los threads que quieren sacar datos si no hay datos disponibles

Ver "[productores-consumidores](#)".

Ver "[BlockingQueue](#)".

Ver "[PipedOutputStream](#)".

5. Concurrencia (concepto)

Se habla de concurrencia o de programas concurrentes cuando el programa se descompone en varias tareas. En este documento nos centraremos en el modelo de concurrencia de los *threads* de java que se caracterizan por

- compartir memoria
- repartirse el tiempo de CPU (tenga el equipo una o más CPUs)

Ver "[thread](#)".

Toda solución concurrente a un problema puede calificarse según las siguientes propiedades:

- safety ([[seguridad]]): no ocurre nada malo;
contrarias a la seguridad son las carreras [race conditions](#)

- liveness ([[vivacidad]]): antes o después, ocurre algo bueno; contrarios a la vivacidad son el *interbloqueo* [deadlock] y el bloqueo activo [[livelock]]
- fairness ([[equidad]]): los recursos se reparten con justicia; contrario a la equidad es la *inanición* [starvation]

6. *Concurrent collections (clases)*

El paquete java.util.concurrent proporciona varios tipos de colecciones especialmente diseñadas para programas concurrentes:

CopyOnWriteArrayList<E>

Una implementación de List<E>

BlockingQueue<E>

Una cola FIFO que bloquea el meter hasta que hay sitio y el sacar hasta que hay contenido.

ConcurrentMap<K, V>

Una estructura asociativa, tipo Map<K, V>

CopyOnWriteArraySet<E>

Una implementación de Set<E>

7. *Condición [condition] (concepto)*

Cola de espera asociada a objetos de tipo Lock.

Ver "*Lock*".

Alrededor de una condición podemos dejar un thread esperando, await(), o reactivarlo, signal().

7.1. Ejemplo

Se trata de una fábrica con varios proveedores. Para cada proveedor hay una fiche que indica si tenemos existencias y cuales son los mínimo y máximos de existencias en el almacén:

class Ficha
<pre>public class Ficha { private final int indice; private final String nombre; private final int min; private final int max; private int existencias; public Ficha(int indice, String nombre, int min, int max, int existencias) { this.indice = indice; this.nombre = nombre; this.min = min; this.max = max; this.existencias = existencias; } public int getIndice() { return indice; } public String getNombre() {</pre>

```

        return nombre;
    }

    public int getMin() {
        return min;
    }

    public int getMax() {
        return max;
    }

    public int getExistencias() {
        return existencias;
    }

    public void incExistencias() {
        this.existencias++;
    }

    public void decExistencias() {
        this.existencias--;
    }
}

```

La fábrica controla el nivel de existencias con una condición por proveedor

class Fabrica

```

import java.util.Random;
import java.util.concurrent.locks.*;

public class Fabrica extends Thread {
    private final Random random = new Random();

    private final Ficha[] fichas;

    private final Lock proveedores;
    private final Condition[] conditions;

    public Fabrica(Ficha[] fichas) {
        this.fichas = fichas;

        proveedores = new ReentrantLock();

        conditions = new Condition[fichas.length];
        for (int i = 0; i < fichas.length; i++)
            conditions[i] = proveedores.newCondition();
    }

    public void run() {
        try {
            for (int i = 0; i < 10; i++) {
                foto();
                int caso = random.nextInt(fichas.length);
                get(caso);
                System.out.println();
            }
        } catch (InterruptedException ignored) {
        }
        System.exit(0);
    }
}

```

```

    }

    private void get(int i) throws InterruptedException {
        Ficha ficha = fichas[i];
        String id = ficha.getNombre();
        System.out.println("get: " + id);
        Condition condition = conditions[i];
        proveedores.lock();
        while (ficha.getExistencias() == 0)
            condition.await();
        ficha.decExistencias();
        if (ficha.getExistencias() < ficha.getMin()) {
            System.out.println("necesito " + id);
            condition.signal();
        }
        proveedores.unlock();
    }

    public void put(int i) throws InterruptedException {
        Ficha ficha = fichas[i];
        String id = ficha.getNombre();
        Condition condition = conditions[i];
        proveedores.lock();
        while (ficha.getExistencias() >= ficha.getMax())
            condition.await();
        System.out.println("put: " + id);
        ficha.incExistencias();
        condition.signal();
        proveedores.unlock();
    }

    private void foto() {
        proveedores.lock();
        for (Ficha ficha : fichas) {
            System.out.print(ficha.getNombre());
            System.out.print(": ");
            System.out.printf("{%d, %d} ",
                ficha.getMin(), ficha.getMax());
            for (int s = 0; s < ficha.getExistencias(); s++)
                System.out.print("[ ] ");
            System.out.println();
        }
        proveedores.unlock();
    }
}

```

Los proveedores fabrican productos y esperan a poder colocarlos en la fábrica:

class Proveedor

```

public class Proveedor extends Thread {
    private final int id;
    private final Fabrica fabrica;

    public Proveedor(int id, Fabrica fabrica) {
        this.id = id;
        this.fabrica = fabrica;
    }

    public void run() {

```

```

        while (true) {
            try {
                int t = (int) (Math.random() * 3);
                Thread.sleep(t * 1000);
                fabrica.put(id);
            } catch (InterruptedException ignored) {
            }
        }
    }
}

```

Y montamos el escenario con una fábrica y varios proveedores:

class Escenario

```

public class Escenario {
    public static void main(String[] args) {
        Ficha[] fichas = new Ficha[3];
        fichas[0] = new Ficha(0, "A", 2, 5, 2);
        fichas[1] = new Ficha(1, "B", 1, 2, 1);
        fichas[2] = new Ficha(2, "C", 2, 2, 2);

        Fabrica fabrica = new Fabrica(fichas);
        fabrica.start();

        for (int i = 0; i < fichas.length; i++) {
            Proveedor proveedor = new Proveedor(i, fabrica);
            proveedor.start();
        }
    }
}

```

8. **Deadlock [interbloqueo] (concepto)**

Ver "[interbloqueo](#)".

9. **Demonio [daemon] (concepto)**

Tipo de thread.

Los thread demonio se caracterizan porque el programa puede terminar aunque alguno de estos threads siga activo.

Hay métodos en la clase Thread que permiten manejar este atributo:

```

void setDaemon(boolean on)
    Setter.

```

```

boolean isDaemon()
    Getter.

```

10. **Ejecutor [Executor] (concepto)**

La forma clásica de lanzar un thread es

```

Runnable x = ...
Thread thread = new Thread(x);
thread.start();

```

Los objetos Executor lo hacen un poco diferente:

```
Executor executor = ...
Runnable x = ...
executor.execute(x);
```

Visto así, no parece que hayamos ganado gran cosa.

La ventaja aparece al considerar que Executor es una interfaz con múltiples implementaciones que permiten controlar la ejecución del thread.

La variedad la aportan los objetos ExecutorService que se crean con la fábrica (factory) de Executors.

Uno de los usos más habituales es limitar el número de threads concurrentes, de forma que si el programa quiere lanzar más threads, los nuevos se guardan en una cola hasta que termine uno de los threads en ejecución.

Ver "[Executor](#)".

Ver "[ExecutorService](#)".

Ver "[Executors](#)".

10.1. Ejemplo

Lanzamos 10 threads, pero limitamos a 4 el número de threads que ejecutan en paralelo:

class Repeater

```
public class Repeater extends Thread {
    private String id;
    private int count;

    public Repeater(String id, int count) {
        this.id = id;
        this.count = count;
    }

    public void run() {
        do {
            System.out.println(id + ": start: " + count);
            count--;
            try {
                Thread.sleep(1000); // 1000ms = 1s
            } catch (InterruptedException ignore) {
            }
        } while (count > 0);
        System.out.println(id + ": end");
    }
}
```

class Repeaters

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class Repeaters {
    private static final int NUMBER_THREADS = 10;
    private static final int POOL_SIZE = 4;

    public static void main(String[] args) {
        ExecutorService pool =
            Executors.newFixedThreadPool(POOL_SIZE);
        for (int i = 0; i < NUMBER_THREADS; i++) {
            String id = String.valueOf((char) ('A' + i));
```

```
        int count = (int) (Math.random() * 5);
        Runnable repeater = new Repeater(id, count);
        pool.execute(repeater);
    }
    System.out.println("Repeaters ...");
}
}
```

> java Repeaters

```
A: start: 0
B: start: 3
C: start: 0
D: start: 3
Repeaters ...
A: end
B: start: 2
E: start: 4
C: end
F: start: 0
D: start: 2
B: start: 1
F: end
G: start: 2
E: start: 3
D: start: 1
B: end
H: start: 4
D: end
I: start: 0
E: start: 2
G: start: 1
H: start: 3
I: end
J: start: 1
G: end
E: start: 1
H: start: 2
J: end
E: end
H: start: 1
H: end
```

11. Equidad [fairness] (concepto)

Se dice de un thread que cuando desea progresar lo hace con justicia respecto de los demás, es decir repartiendo recursos.

Ver "inanición".

12. Excepciones [exceptions]

Un thread puede lanzar una excepción y capturarla o no. Si el thread no la captura, la excepción sale fuera del método run().

A un thread le podemos asociar un manejador de excepciones no capturadas, manejador que se llama cuando una excepción se sale del método run() sin haber sido capturada.

```
class Problematica
```

```

class Problematica extends Thread {
    private final int n;

    public Problematica(int n) {
        this.n = n;
    }

    public void run() {
        int resultado = 100 / n;
    }
}

```

class Problemas

```

import java.lang.Thread.UncaughtExceptionHandler;

public class Problemas {
    public static void main(String[] args) {
        UncaughtExceptionHandler eh = new Manejador();
        Problematica p = new Problematica(0);
        p.setName("tarea problemática");
        p.setUncaughtExceptionHandler(eh);
        p.start();
    }
}

```

class Manejador

```

import java.lang.Thread.UncaughtExceptionHandler;

class Manejador implements UncaughtExceptionHandler {
    public void uncaughtException(Thread thread, Throwable e) {
        System.err.println(thread.getName() + ": " + e);
    }
}

```

> java Problemas

```

tarea problemática: java.lang.ArithmeticException: / by zero

```

13. Fairness [equidad] (concepto)

Ver "[equidad](#)".

14. Hebra [thread] (concepto)

Ver "[thread](#)".

15. Hilo [thread] (concepto)

Ver "[thread](#)".

16. IllegalMonitorStateException [excepción] java.lang...

Excepción que se lanza cuando se intenta hacer una operación propia de un monitor fuera de un monitor. En otras palabras, el thread debe poseer el cerrojo sobre el que intenta hacer una operación wait() o notify().

Para ejecutar `wait()` o `notify()` debemos estar dentro de un método o una zona `synchronized`, Podemos estarejecutando el cuerpo del método, o podemos estar en otro código que se llama desde el cuerpo del método.

En métodos sincronizados, sólo podemos hacer `wait()` o `notify()` sobre el objeto que ejecuta el método

```
synchronized metodo() {  
    ...  
    this.wait();  
    ...  
    this.notify();  
    ...  
}
```

En zonas `synchronized`, sólo podemos hacer `wait()` o `notify` sobre el cerrojo que controla la zona:

```
synchronized (cerrojo) {  
    ...  
    cerrojo.wait();  
    ...  
    cerrojo.notify();  
    ...  
}
```

17. Inanición [*starvation*] (concepto)

Se dice que un programa sufre inanición cuando algún `thread` es incapaz de disfrutar de los recursos que necesita porque otros `threads` no los liberan. Aunque el programa sigue haciendo cosas, el `thread` afectado no va a ninguna parte.

Ver "[equidad](#)".

18. Interbloqueo [*deadlock*] (concepto)

Se da una situación de "deadlock" cuando dos o más `threads` se se bloquean mutuamente, esperando el uno al otro, sin progresar ninguno de ellos.

Aunque parezca obvio el error, es frecuente provocar un deadlock cuando dos o más `threads` intentan ganar acceso a dos o más zonas de exclusión mutua, pero entrelazan los cerrojos:

<pre>synchronized(A) { synchronized(B) { ... } }</pre>	<pre>synchronized(B) { synchronized(A) { ... } }</pre>
--	--

Ver "[vivacidad](#)".

19. Interrupciones (concepto)

Un `thread` puede interrumpir a otro:

```
otra.interrupt();
```

El `thread` interrumpido lo ve como una excepción si está esperando algo, o tiene que mirarlo explícitamente si está corriendo.

Para lo que sigue, supondremos que la llamada `interrupt()` activa un flag en la tarea interrumpida:

```
private boolean flagI = false;
flagI = true;
```

La tarea interrumpida puede hacer 3 cosas:

1. Recibir una `InterruptedException`
2. Chequear con `interrupted()`
3. Chequear con `isInterrupted()`

Cada una de las tres opciones es sutilmente diferente.

La primera opción es recibir una **`InterruptedException`**. Esto ocurre cuando la tarea está realizando alguna operación que la tiene bloqueada. Por ejemplo

```
try {
    Thread.sleep(s * 1000);
} catch (InterruptedException e) {
    // habrá que hacer algo
}
```

Java captura la excepción y, además, resetea el flag

```
flagI = false;
```

Si el thread no está esperando, entonces sigue funcionando normalmente, pero podemos comprobar de vez en cuando si hay una interrupción pendiente.

Tenemos dos formas de saber si el thread tiene una solicitud de interrupción pendiente de atender:

```
static boolean interrupted()
```

Es una llamada a la clase `Thread`. Devuelve `TRUE` si el thread que se está ejecutando tiene una interrupción pendiente.

Y según responde, elimina la interrupción.

```
flagI = false;
```

```
boolean isInterrupted()
```

Es una llamada a un objeto `Thread`. Devuelve `TRUE` si el objeto tiene una interrupción pendiente.

A diferencia del otro método, la interrupción permanece pendiente: No se modifica `flagI`. Como consecuencia, en algún momento habrá que llamar a `interrupted()` para resetearlo.

si ...	flagI
<code>interrupt()</code>	<code>flagI = true;</code>
<code>catch (InterruptedException e)</code>	<code>flagI = false;</code>
<code>if (interrupted())</code>	<code>flagI = false;</code>
<code>if (isInterrupted ())</code>	<code>// sigue igual</code>

19.1. Ejemplo

El siguiente ejemplo usa el mecanismo de interrupción para terminar un thread:

- regularmente muestrea si está interrumpido y, si es así, se acaba
- los métodos que bloquean la ejecución, se pueden ver interrumpidos
- el método para cancelar lanza una interrupción que es recogida bien por el bucle, bien por el manejador (catch)

class Interrumpible

```
public class Interrumpible
    extends Thread {
    public void run() {
        try {
            while (!isInterrupted()) {
                System.out.println("mis tareas ...");
                Thread.sleep(1000);
            }
        } catch (InterruptedException ignored) {
        }
        System.out.println("tareas de terminacion");
    }

    public void cancelar() {
        interrupt();
    }
}
```

20. interrupt (método) void interrupt()

Método de los objetos tipo Thread. Se usa para mandarle una señal a una thread.

Ver "[interrupciones](#)".

21. interrupted (método) static boolean interrupted()

Método de la clase Thread. Devuelve true si el thread ha sido objeto de una interrupción.

Al llamar a este método la señal desaparece; es decir, si se le llama dos veces seguidas, la primera vez devuelve TRUE y la segunda vez devuelve FALSE .

Ver "[interrupciones](#)".

22. InterruptedException (excepción)

```
class java.lang.InterruptedException
    extends Exception
```

Si una tarea quiere despertar otra, le manda una interrupción

```
otra.interrupt();
```

Si la otra tarea está activa, lo único que pasa es que se marca el flag de interrupción.

Pero si la otra tarea está bloqueada esperando a que ocurra algo, en lugar de marcar el flag, se le lanza una InterruptedException. El efecto es que la tarea se despierta y sabe que es porque la ha interrumpido otra.

Si no hacemos nada, la interrupción se limita a despertar a la tarea.

A veces lo que se hace es aprovechar el manejador para activar el flag y centralizar en otro sitio el manejo de la interrupción:

```

try {
    ...;
} catch (InterruptedException e) {
    interrupt();
}

```

Ver "[interrupciones](#)".

23. *isInterrupted* (método) boolean *isInterrupted()*

Método de los objetos tipo Thread para saber si el thread ha sido objeto de una interrupción.

Ver "[interrupciones](#)".

24. *join* (método) void *join()*

Método de la clase "java.lang.Thread" que bloquea al thread llamante hasta que el thread llamado acabe de ejecutar su método run().

class Repeater

```

public class Repeater extends Thread {
    private String id;
    private int count;

    public Repeater(String id, int count) {
        this.id = id;
        this.count = count;
    }

    public void run() {
        do {
            System.out.println(id + ": " + count);
            count--;
            try {
                Thread.sleep(1000); // 1000ms = 1s
            } catch (InterruptedException ignore) {
            }
        } while (count > 0);
        System.out.println(id + ": end");
    }
}

```

class WaitingRepeaters

```

public class WaitingRepeaters {
    public static void main(String[] args)
        throws InterruptedException {
        Repeater threadA = new Repeater("A", 2);
        Repeater threadB = new Repeater("B", 5);

        threadA.start();
        threadB.start();

        System.out.println("Repeaters ...");
        threadA.join();
        System.out.println("A joins");
    }
}

```

> java WaitingRepeaters

A: 2

```
Repeaters ...
B: 5
A: 1
B: 4
A: end
B: 3
A joins
B: 2
B: 1
B: end
```

25. Livelock (concepto)

A veces se denomina “interbloqueo activo” o “estancamiento”. En inglés, “catch-22”.

Se da una situación de “livelock” cuando un thread A está trabajando para un thread B, que a su vez está trabajando para el thread A, y así sucesivamente. Es una especie de “recursividad sin fin”. Los threads están activos, pero tanto trabajo no lleva a un resultado productivo.

Ver “*liveness*”.

26. Liveness [vivacidad] (concepto)

Se dice que un programa disfruta de “vivacidad” cuando responde ágilmente a las solicitudes del usuario. Cuando algo bueno ocurre, más bien antes que después.

Hay varios problemas que pudieran socavar esta propiedad:

- “*deadlock*” (interbloqueo por inactividad)
- “*livelock*” (bloqueo por hiperactividad)

27. Lock [cerrojo] (concepto)

Ver “*cerrojos*”.

28. Monitor (concepto)

Es un componente clásico de lenguajes de programación con control de concurrencia.

En terminología java, sería una clase tal que

- los campos son privados y sólo accesibles por métodos de acceso
- todos los métodos de acceso están sincronizados
- Los threads que intentan acceder al monitor quedan bloqueados (wait) hasta que se les despierta (notify() o notifyAll())

Se puede decir que java generaliza el concepto de monitor al no imponer todas las condiciones mencionadas a la vez; es decir, se puede programar una clase como un monitor pero también se pueden programar clases de forma más flexible.

Ver “*zonas de exclusión mutua*”.

29. Mutex (concepto)

Ver “*zonas de exclusión mutua*”.

30. *notify (método) void notify()*

Método de la clase java.lang.Object.

Cuando un thread está en una zona de acceso sincronizado, al llamar a wait despierta al primer thread que haya esperando (wait) para acceder a dicha zona.

Ver "wait()". Ver "notifyAll()".

notify() es más eficiente que notifyAll() pues sólo despierta 1 thread; pero es necesario recurrir a notifyAll() cuando en la cola puede haber diferentes threads esperando a diferentes condiciones.

31. *notifyAll (método) void notifyAll()*

Método de la clase java.lang.Object.

Cuando un thread está en una zona de acceso sincronizado, al llamar a wait despierta a todos los threads que haya esperando (wait) para acceder a dicha zona.

Ver "wait()". Ver "notifyAll()".

notify() es más eficiente que notifyAll() pues sólo despierta 1 thread; pero es necesario recurrir a notifyAll() cuando en la cola puede haber diferentes threads esperando a diferentes condiciones.

32. *Operación atómica (concepto)*

Se dice que una operación es atómica cuando se completa de principio a fin sin interrupciones.

Java garantiza que son atómicos los accesos a las variables de tipos primitivos, excepto double y long. Ver "variables atómicas".

Java garantiza que son atómicos los accesos a todas las variables de tipos primitivos etiquetadas como "volatile".

Podemos decir que las zonas y métodos synchronized demarcan operaciones atómicas complejas.

Ver "zonas de exclusión mutua".

33. *Race conditions [carreras] (concepto)*

Se dice que un programa sufre este problema cuando el resultado puede ser correcto o incorrecto dependiendo de cómo se de la casualidad de que se entrelacen los threads.

Básicamente el problema aparece cuando varios threads hacen ciclos de modificación de una variable compartida (lectura + modificación + escritura) y el ciclo completo se ve interrumpido

Ejemplo:

	thread 1 { x = x+5; }	thread 2 { x = x+100; }
t0	x vale 100	
t1	lee 100	
t2	incrementa a 105	
t3		lee 100
t4	escribe 105	

t5		incrementa a 200
t6		escribe 200
final	x vale 200	

Es el problema principal que hace que los programas sean incorrectos.

Es un problema muy difícil de detectar, pues el programa puede funcionar bien millones de veces y de repente fallar porque se da una coincidencia imprevista. Es por ello que mucho esfuerzo de la programación concurrente se dedica a construir el programa de forma que no puedan haber carreras. Eso suele exigir pruebas formales de seguridad.

34. **Reentrante [reentrant] (concepto)**

Se dice que un código es reentrante cuando puede ser utilizado concurrentemente por varios threads sin interferencias entre ellos.

Es sinónimo de "*thread-safe*".

Ver "*seguridad*".

35. **run (método) void run()**

Método de la interfaz Runnable.

```
void run()
```

Es el que se ejecuta cuando un objeto Runnable se usa en un thread y este arranca

```
Runnable x = ...
Thread thread = new Thread(x);
thread.start(); // ejecuta run()
```

La clase Thread implementa Runnable, de forma que al definir un thread podemos sobrescribir el método run() y podremos arrancar directamente la tarea:

```
Thread thread = ...
thread.start(); // ejecuta run()
```

36. **Safety [seguridad] (concepto)**

Ver "*seguridad*".

37. **Sección crítica**

Ver "*zonas de exclusión mutua*".

38. **Seguridad [safety] (concepto)**

Se dice que un programa es seguro cuando no genera nunca resultados incorrectos.

Se dice que un programa concurrente viola el principio de seguridad cuando se generan datos incorrectos.

Dado que los diferentes threads pueden ejecutarse concurrentemente de muchas maneras, se dice que un programa es seguro cuando ninguna de las maneras genera resultados erróneos.

Ver "*thread-safe*".

Ver "*race conditions*".

En programación concurrente se estudian muchos mecanismos para evitar comportamientos incorrectos:

- cerrojos [locks]
- zonas de exclusión mutua
- semáforos [semaphores]
- monitores [monitors]

39. Semáforo [*semaphore*] (concepto)

Es uno de los sincronizadores más clásicos para establecer zonas de exclusión mutua.

Los semáforos más sencillos son binarios. Para entrar en una zona crítica un thread debe adquirir el derecho de acceso, y al salir lo libera.

```
Semaphore semaphore = new Semaphore(1);

semaphore.acquire();
// zona crítica
semaphore.release();
```

Por solidez, nunca debemos olvidar liberar un semáforo al salir de la zona crítica. El código previo lo hace si salimos normalmente; pero si se sale de forma abrupta (return o excepción) entonces el semáforo no se liberaría. Es por ello, que normalmente se sigue este patrón:

```
semaphore.acquire();
try {
    // zona crítica
} finally {
    semaphore.release();
}
```

Más general, el semáforo puede llevar cuenta de N permisos. Los threads solicitan algunos permisos; si los hay, los retiran y siguen; si no los hay, quedan esperando a que los haya. Cuando ha terminado, el thread devuelve los permisos.

Ver "*Semaphore*".

Ver "*zonas de exclusión mutua*"

39.1. Ejemplo: semáforo binario

class ContadorCompartido

```
public class ContadorCompartido {
    private int n = 0;

    public int getN(String id) {
        return n;
    }

    public void setN(String id, int n) {
        this.n = n;
        System.err.println(id + ": " + n);
    }
}
```

class IncrementadorLento

```
import java.util.concurrent.Semaphore;
```

```

public class IncrementadorLento extends Thread {
    private final String id;
    private final ContadorCompartido cc;

    private static Semaphore semaforo = new Semaphore(1);

    public IncrementadorLento(String id, ContadorCompartido cc) {
        this.id = id;
        this.cc = cc;
    }

    @Override
    public void run() {
        try {
            semaforo.acquire();
        } catch (InterruptedException e) {
            System.err.println(id + ": " + e);
        }

        try {
            int valor = cc.getN(id);
            valor++;
            sleep(1000);
            cc.setN(id, valor);
        } catch (InterruptedException e) {
            System.err.println(id + ": " + e);
        } finally {
            semaforo.release();
        }
    }
}

```

39.2. Ejemplo: semáforo con N permisos

class Tarea

```

import java.util.concurrent.Semaphore;

public class Tarea extends Thread {
    private Semaphore contador;

    public Tarea(Semaphore contador) {
        this.contador = contador;
    }

    public void run() {
        // hace su tarea
        contador.release();
    }
}

```

class EsperaNThreads

```

public class EsperaNThreads {
    public static void main(String[] args)
        throws InterruptedException {
        Semaphore contador = new Semaphore(0);
        List<Tarea> tareas = new ArrayList<Tarea>();
        tareas.add(new Tarea(contador));
        // ... N veces
    }
}

```

```
    for (Tarea tarea : tareas)
        tarea.start();

    // espera a que todas acaben
    contador.acquire(tareas.size());
}
}
```

40. Sincronizadores (concepto)

Clases que permiten montar rápidamente arquitecturas típicas de coordinación de threads. Java proporciona los siguientes:

Semaphore (semáforos)

Es uno de los sincronizadores más clásicos. Se trata de llevar cuenta de N permisos. Los threads solicitan permisos; si los hay, los retiran y siguen; si no los hay, quedan esperando a que los haya. Cuando ha terminado, el thread devuelve los permisos.

Ver "[semáforo](#)".

Ver "[Semaphore](#)".

CountDownLatch

Se establece un contador. Los threads se registran y esperan a que termine la cuenta atrás. El contador va decrementándose y cuando llega a cero, todos los threads registrados se reactivan.

Ver "[CountDownLatch](#)".

CyclicBarrier

Similar a CountDownLatch, con la diferencia de que cuando llega a cero, se resetea a la cuenta inicial.

Ver "[CyclicBarrier](#)".

Exchanger

Define un punto de encuentro en el que dos threads se encuentran e intercambian un objeto entre ellos.

Ver "[Exchanger](#)".

41. sleep (método) void sleep(long)

Métodos de la clase java.lang.Thread:

```
static void sleep(long millis)
    throws InterruptedException
static void sleep(long millis, int nanos)
    throws InterruptedException
```

El método sleep(t) de la clase Thread permite detener la ejecución del thread durante t milisegundos.

La forma más habitual de usarlo es

```
try {
    Thread.sleep(ms);
} catch (InterruptedException e) {
    ...
}
```

```
}
```

sleep() lanza una interrupción si este thread se ve interrumpido por otro durante la espera.

42. start (método) void start()

Método de la clase java.lang.Thread que se emplea para arrancar un thread.

Ver "[thread](#)".

43. Starvation [inanición] (concepto)

Ver "[inanición](#)".

44. stop (deprecated) void stop()

Metodo que se usaba para detener un thread. Se recomienda no usarlo pues no hay control de en qué punto se detiene el thread.

Es mejor parar los threads por medio de un campo de solicitud de detención, o interrumpiéndolos.

Ver "[thread](#)". Ver "[interrupciones](#)".

45. synchronized (palabra reservada)

Se utiliza para demarcar zonas de exclusión mutua. Se utiliza de 2 formas: marcando un bloque de sentencias o marcando métodos enteros.

Zonas de exclusión mutua

Sirve para delimitar zonas de exclusión mutua: sólo un thread puede estar ejecutando código de la zona en un momento dado.

```
synchronized (objeto) {  
    // zona de exclusión mútua;  
}
```

Se puede usar cualquier objeto java como control, aunque se recomienda utilizar objetos privados para controlar quien controla la zona. Un objeto público podría usarse desde el exterior para bloquear la zona.

Métodos sincronizados

Si en una clase varios métodos se marcan como sincronizados, sólo se permitirá 1 thread accediendo a alguno de los métodos marcados en cada momento

```
public class ... {  
    public synchronized ... metodo1(...) { ... }  
    public synchronized ... metodo2(...) { ... }  
}
```

Sólo 1 thread puede acceder en cada momento al método 1 o al método 2.

Técnicamente, cada método lo que hace es rodear su cuerpo de sentencias con una sincronización sobre el propio objeto:

```
public ... metodo1(...)  
    synchronized(this) {  
        ...  
    }  
}
```

Ver “monitores”.

Terminación abrupta

Una zona sincronizada puede terminar abruptamente por diferentes motivos, siendo lo más frecuentes que salte una EXCEPTION que se captura fuera o que se acaba con un RETURN. En cualquier caso, java se encarga de abrir el cerrojo.

El siguiente ejemplo tiene métodos sincronizados, que terminan abruptamente cuando la cantidad a meter o sacar es negativa, o cuando no hay saldo suficiente:

```
public class CuentaCorriente {
    private int saldo = 0;

    public synchronized int meter(int cantidad) {
        if (cantidad < 0)
            return saldo;
        saldo += cantidad;
        return saldo;
    }

    public synchronized int sacar(int cantidad)
        throws Exception {
        if (cantidad < 0)
            return saldo;
        if (saldo < cantidad)
            throw new Exception("no hay saldo suficiente");

        saldo -= cantidad;
        return saldo;
    }
}
```

Variables volatile en zonas synchronized

Ver “*volatile en synchronized*”

45.2. Ejemplo: bloque de sentencias sincronizado

class ContadorCompartido

<pre>public class ContadorCompartido { private int n = 0; public int getN(String id) { return n; } public void setN(String id, int n) { this.n = n; System.err.println(id + ": " + n); } }</pre>
--

class IncrementadorLento

<pre>public class IncrementadorLento extends Thread { private final String id; private final ContadorCompartido cc; public IncrementadorLento(String id, ContadorCompartido cc) { this.id = id; } }</pre>
--

```

        this.cc = cc;
    }

    @Override
    public void run() {
        try {
            synchronized (cc) {
                int valor = cc.getN(id);
                valor++;
                sleep(1000);
                cc.setN(id, valor);
            }
        } catch (Exception e) {
            System.err.println(e);
        }
    }
}

```

45.3. Ejemplo: cuenta corriente

Como ejemplo, sea esta cuenta absurda que mira el saldo y espera un ratito antes de actualizarlo en cuenta:

class Cuenta

```

public class Cuenta {
    private volatile int saldo = 0;

    public int getSaldo() {
        return saldo;
    }

    public int ingresar(int dinero) {
        try {
            int x = saldo + dinero;
            int s = (int) (Math.random() * 2000);
            Thread.sleep(s);
            saldo = x;
        } catch (InterruptedException e) {
            System.err.println(e);
        }
        return saldo;
    }

    public int sacar(int dinero) {
        try {
            int x = saldo - dinero;
            int s = (int) (Math.random() * 2000);
            Thread.sleep(s);
            saldo = x;
        } catch (InterruptedException e) {
            System.err.println(e);
        }
        return saldo;
    }
}

```

Si ponemos a funcionar varios clientes a la vez, tenemos problemas de actualización de la variable “saldo”.

A continuación se resentan dos soluciones, ambas consistentes en convertir la cuenta compartida en un monitor.

Solución con zonas de exclusión mutua

Para controlar la actualización de la variable “saldo” podemos establecer una zona de exclusión mutua usando cualquier objeto. En el código que sigue se han establecido 2 zonas de exclusión, usando ambas el mismo objeto de control, de forma que la exclusión abarca ambas zonas:

class Cuenta modificada

```
public class Cuenta {
    private volatile int saldo = 0;
    private final Object zona = new Object();

    public int getSaldo() {
        synchronized (zona) {
            return saldo;
        }
    }

    public int ingresar(int dinero) {
        synchronized (zona) {
            try {
                int x = saldo + dinero;
                int s = (int) (Math.random() * 2000);
                Thread.sleep(s);
                saldo = x;
            } catch (InterruptedException e) {
                System.err.println(e);
            }
            return saldo;
        }
    }

    public int sacar(int dinero) {
        synchronized (zona) {
            try {
                int x = saldo - dinero;
                int s = (int) (Math.random() * 2000);
                Thread.sleep(s);
                saldo = x;
            } catch (InterruptedException e) {
                System.err.println(e);
            }
            return saldo;
        }
    }
}
```

Solución con métodos sincronizados

Para controlar la actualización de la variable “saldo” podemos marcar todos los métodos peligrosos. En el código que sigue se han marcado 3 métodos, de forma que la exclusión abarca a todos conjuntamente:

class Cuenta modificada

```
public class Cuenta {
    private volatile int saldo = 0;
```

```

public synchronized int getSaldo() {
    return saldo;
}

public synchronized int ingresar(int dinero) {
    try {
        int x = saldo + dinero;
        int s = (int) (Math.random() * 2000);
        Thread.sleep(s);
        saldo = x;
    } catch (InterruptedException e) {
        System.err.println(e);
    }
    return saldo;
}

public synchronized int sacar(int dinero) {
    try {
        int x = saldo - dinero;
        int s = (int) (Math.random() * 2000);
        Thread.sleep(s);
        saldo = x;
    } catch (InterruptedException e) {
        System.err.println(e);
    }
    return saldo;
}
}

```

Ver “monitores”.

46. Thread [hilo] (concepto)

A veces se traduce por “hebra”, “hilo”, etc.

Los programas más normales ejecutan una secuencia de instrucciones desde que arrancan hasta que terminan.

Los programas multi-thread disponen de varios threads o secuencias de instrucciones que se ejecutan en paralelo, desde que cada uno arranca hasta que terminan.

Ejecutar en paralelo puede significar varias cosas:

- en un sistema con 1 CPU, se reparten el tiempo de la CPU: “un ratito cada thread”
- en un sistema con varias CPU, se pueden asignar diferentes secuencias a diferentes CPUs
- si hay más threads que CPUs, un poco de ambas formas

Un programa java tiene un thread principal con el que arranca. Por programa podemos arrancar más threads.

Necesitamos objetos que implementen la interfaz Runnable. Hay dos formas habituales:

- se implementa la interfaz Runnable
- se extiende la clase Thread (que a su vez es implementa Runnable)

¿Cómo se crea un thread? Opción 1: new Thread(Runnable)

class Repeater

```
public class Repeater implements Runnable {
    private String id;
    private int count;

    public Repeater(String id, int count) {
        this.id = id;
        this.count = count;
    }

    public void run() {
        do {
            System.out.println(id + ": " + count);
            count--;
            try {
                Thread.sleep(1000); // 1000ms = 1s
            } catch (InterruptedException ignore) {
            }
        } while (count > 0);
        System.out.println(id + ": end");
    }
}
```

Una vez dispone de objetos que implementan la interfaz Runnable, cree el Thread:

```
Runnable repeater = new Repeater(id, count);
Thread thread = new Thread(repeater);
```

¿Cómo se crea un thread? Opción 2: extends Thread

class Repeater

```
public class Repeater extends Thread {
    private String id;
    private int count;

    public Repeater(String id, int count) {
        this.id = id;
        this.count = count;
    }

    public void run() {
        do {
            System.out.println(id + ": " + count);
            count--;
            try {
                Thread.sleep(1000); // 1000ms = 1s
            } catch (InterruptedException ignore) {
            }
        } while (count > 0);
        System.out.println(id + ": end");
    }
}
```

Y creamos threads directamente:

```
Thread thread = new Repeater(id, count);
```

¿Cómo se arranca un thread?

Ejecutando el método `run()`.

Ver "[ejecutor](#)".

A continuación, hay 2 threads operando: el original y el recién arrancado. El siguiente método arranca 3 threads y sigue, así que tenemos 4 threads concurrentes:

```
public static void main(String[] args) {
    for (int i = 0; i < 3; i++) {
        String id = String.valueOf((char) ('A' + i));
        int count = (int) (Math.random() * 10);
        Thread thread = ...;
        thread.start();
    }
    System.out.println("Repeaters ...");
}
```

Cuando se ejecuta este código, vemos algo así:

```
> java Repeaters
```

```
A: 2
B: 3
C: 2
Repeaters ...
A: 1
B: 2
C: 1
A: end
B: 1
C: end
B: end
```

¿Cuándo termina un thread?

Cuando se acaba el código que tiene que ejecutar. O sea, cuando llega al final del método `run()`.

Si es un demonio, o cuando llega al final de `run()` o cuando acaba el programa.

¿Cuándo acaba el programa?

El programa acaba cuando han acabado todos sus threads normales. Son normales los threads que nos son demonios.

Un programa termina aunque haya threads demonio funcionando. Estos demonios abortan.

¿Cómo se para un thread?

Si queremos que un thread se detenga, necesitamos influir en su código para que salte al final y termine.

Es habitual utilizar un campo de la clase para marcar la condición "tienes que parar". La propia clase decide cual es el mejor momento para parar. A veces se denomina "parada ordenada".

Otra opción es interrumpir el thread. Ver "[interrupciones](#)".

```
class StoppableRepeater
```

```

public class StoppableRepeater extends Thread {
    private String id;
    private int count;

    private volatile boolean stop= false;

    public StoppableRepeater(String id) {
        this.id = id;
        this.count = 0;
    }

    public void setStop(boolean stop) {
        this.stop = stop;
    }

    public void run() {
        do {
            System.out.println(id + ": " + count);
            count++;
            try {
                Thread.sleep(1000); // 1000ms = 1s
            } catch (InterruptedException ignore) {
            }
        } while (!stop);
        System.out.println(id + ": end");
    }
}

```

class StoppableRepeaters

```

public class StoppableRepeaters {
    public static void main(String[] args) {
        StoppableRepeater threadA = new StoppableRepeater("A");
        StoppableRepeater threadB = new StoppableRepeater("B");

        threadA.start();
        threadB.start();

        System.out.println("Repeaters ...");
        sleep(2);
        threadA.setStop(true);
        sleep(2);
        threadB.setStop(true);
    }

    private static void sleep(int s) {
        try {
            Thread.sleep(s * 1000);
        } catch (InterruptedException ignore) {
        }
    }
}

```

> java StoppableRepeaters

```

A: 0
Repeaters ...
B: 0
A: 1
B: 1
A: end
B: 2

```

```
B: 3
B: end
```

¿Cómo espera un thread a que otro termine?

Ver "[join\(\)](#)".

Ver "[sincronizadores](#)".

¿Cómo se le pasan valores a un thread?

Como a cualquier objeto:

- en el constructor
- llamando a sus métodos (ver [métodos sincronizados](#))

47. Thread-safe (concepto)

Se dice que una clase es thread-safe cuando los objetos de dicha clase se pueden utilizar desde varios threads sin violar el principio de seguridad.

Se dice que el código es reentrante.

Ver "[seguridad](#)" y "[race conditions](#)".

48. Variables atómicas

El paquete `java.util.concurrent.atomic` define variables en las que todos los métodos están sincronizados. Por ejemplo

- [AtomicBoolean](#)
- [AtomicInteger](#)
- [AtomicIntegerArray](#)
- [AtomicLong](#)
- [AtomicLongArray](#)

Las siguientes clases (tipo monitor) son equivalentes (aunque probablemente el uso de `AtomicInteger` sea algo más rápido en ejecución):

class SynchronizedCounter	class AtomicCounter
<pre>public class SynchronizedCounter { private int c = 0; public synchronized void increment() { c++; } public synchronized void decrement() { c--; } public synchronized int value() { return c; } }</pre>	<pre>import java.util.concurrent .atomic.AtomicInteger; public class AtomicCounter { private AtomicInteger c = new AtomicInteger(0); public void increment() { c.incrementAndGet(); } public void decrement() { c.decrementAndGet(); } public int value() { return c.get(); } }</pre>

	}
--	---

49. Vivacidad [*liveness*] (concepto)

Ver "*liveness*".

50. *volatile* (palabra reservada)

Es un atributo para los campos de un objeto y las variables de un método. Es un aviso al compilador:

Esta variable la utilizan varios threads a la vez; no se permite ninguna optimización del código ejecutable. En particular, si hay varias CPUs, ni se te ocurra hacer copias locales para acceder más deprisa.

Cuando se emplea, java se encarga de garantizar que siempre se accede al último valor asignado a la variable, y que las operaciones de escritura y lectura de la variable serán atómicas.

Declarar una variable como volátil es una forma sencilla de conseguir el mismo efecto que protegiendo mediante *synchronized* todos los accesos a la misma.

volatile* en *synchronized

Java garantiza que dentro de una *zona sincronizada*, todas las variables se tratan como *volatile*; es decir, se leen y escriben en memoria RAM.

En otras palabras, cuando una variable compartida por varios threads siempre se maneja dentro de *zonas sincronizadas*, podemos obviar el atributo *volatile*.

51. *wait()* (método)

Método de la clase `java.lang.Object`.

Cuando un thread está dentro de una zona de acceso sincronizado y llama a `wait()`:

1. libera el acceso a la zona, permitiendo que otro thread entre
2. se coloca en una cola de espera de la que saldrá cuando otro thread, en la misma zona de acceso exclusivo, haga "notify()" o "notifyAll()".

Ver "*productores-consumidores*".

Ver "*dining philosophers*".

Ver "*readers & writers*".

Existe una variante de "wait()" que admite un argumento que son los segundos que el thread está dispuesto a esperar. Pasado ese tiempo, se lanza una excepción que aborta la espera.

51.1. **while(condición) { wait(); }**

Cuando llamar a `wait()` depende de alguna condición, prácticamente siempre las llamadas a `wait()` se producen dentro de un bucle `while`

```
while (condicion()) {
    wait();
}
```

Esto es así para que el thread se detenga siempre que se de la condición, tanto si es la primera vez, como si hemos salido del `wait()` por alguna razón.

Una alternativa sugerente, pero errónea, sería

```
if (condicion()){
    wait();
}
```

Que sólo chequea la condición la primera vez. Esto está mal.

- Normalmente se sale del `wait()` porque otro thread hace un `notify()` o un `notifyAll()`. No podemos evitar el `while` salvo que sea posible demostrar que el `notify()` se ejecuta tras verificar que la condición es cierta y que si varios threads despiertan, es cierto para todos ellos. En general, puede ser difícil de demostrar.
- Incluso si podemos llegar a demostrar que un `notify()` garantiza que la condición se cumple, java no garantiza que el thread que despierta se ejecute inmediatamente: simplemente lo marca como “puede ejecutar”. Para cuando el thread ejecuta, la condición puede seguir siendo cierta, o puede que ya no lo sea. Eso puede ser difícil de garantizar.
- Java no garantiza que un thread sólo pueda salir de una espera con un `notify()`. Un thread puede despertarse de forma espúrea. En estos casos, o volvemos a verificar la condición o el thread se pondría en movimiento incorrectamente. Esto depende más del sistema operativo que hay por debajo de java que de java propiamente dicho.

52. *yield()* (método) *void yield()*

Es un método de los threads que permite indicarle al *hardware* que está ejecutando la tarea, que puede interrumpirla y darle una oportunidad a otro thread.

En líneas generales, se recomienda no usar este método nunca.

No está muy clara su semántica. Una implementación trivial es no hacer nada, es decir es como si el thread le da una oportunidad a la CPU para que elija otro thread, pero la CPU vuelve a elegir el mismo.

Puede pensarse que cuando un thread llama a `yield()` la CPU intenta llamar a otro thread y que conseguiremos cierta equidad en la ejecución, aunque nadie lo garantiza.

Es de desear que el algoritmo para elegir qué thread ejecuta es equitativo y va repartiendo juego entre todos los threads listos para ejecutar.

52.1. Ejemplo

El siguiente ejemplo muestra threads que llaman a `yield()` en cada pasada por el bucle. La ejecución muestra que, en el ordenador donde se ha ejecutado este ejemplo, la CPU practica una cierta equidad entre los threads disponibles para ejecutar.

class YieldingThread

```
public class YieldingThread extends Thread {
    private int contador = 3;

    public YieldingThread(int id) {
        super(String.valueOf(id));
    }

    public String toString() {
        return "thread " + getName() + ": " + contador;
    }

    public void run() {
        while (true) {
            System.out.println(this);
            if (contador-- == 0)
                return;
        }
    }
}
```

```

        return;
        yield();
    }
}

public static void main(String[] args) {
    for (int id = 0; id < 5; id++) {
        YieldingThread yt = new YieldingThread(id);
        yt.start();
    }
}
}

```

> java YieldingThread

```

thread 0: 3
thread 1: 3
thread 2: 3
thread 3: 3
thread 4: 3
thread 0: 2
thread 1: 2
thread 2: 2
thread 3: 2
thread 4: 2
thread 0: 1
thread 1: 1
thread 2: 1
thread 3: 1
thread 4: 1
thread 0: 0
thread 1: 0
thread 2: 0
thread 3: 0
thread 4: 0

```

53. Zona crítica (concepto)

Ver "[zonas de exclusión mutua](#)".

54. Zonas de exclusión mutua (concepto)

Se denominan así a partes del código en las que se controla que sólo lo ejecuta un thread en un momento dado. El objetivo es impedir que dos o más ejecuciones pudieran solaparse introduciendo ambigüedad en el resultado final. El problema más habitual es que dos threads modifiquen en paralelo una misma variable con resultado incierto (ver "[race conditions](#)").

Una zona de exclusión significa que java permite la entrada de un solo thread en cada momento. Si un thread llega a una de estas zonas y hay otro thread dentro, el nuevo thread se queda esperando. En un momento dado pueden haber varios threads esperando. Cuando el thread que ocupa la zona sale, uno de los threads que esperan entra.

Hay múltiples formas de establecer zonas:

- formas no estructuradas
 - cerrojos (locks)
 - semáforos
- formas estructuradas
 - bloques synchronized

- métodos synchronized
- monitores

54.1. Ejemplo

Como ejemplo usaremos un contador compartido. Si este código se ejecutara sin control, tendríamos que un thread recoge el valor y antes de devolverlo modificado, otro thread se lo lleva y el resultado es un desastre.

class ContadorCompartido

```
public class ContadorCompartido {
    private int n = 0;

    public int getN(String id) {
        return n;
    }

    public void setN(String id, int n) {
        this.n = n;
        System.err.println(id + ": " + n);
    }
}
```

class IncrementadorLento

```
public class IncrementadorLento extends Thread {
    private String id;
    private ContadorCompartido cc;

    public IncrementadorLento(String id, ContadorCompartido cc) {
        this.id = id;
        this.cc = cc;
    }

    @Override
    public void run() {
        try {
            int valor = cc.getN(id);
            valor++;
            sleep(1000);
            cc.setN(id, valor);
        } catch (InterruptedException e) {
            System.err.println(e);
        }
    }
}
```

Las Bibliotecas de Java

Java proporciona un amplio conjunto de clases para agilizar threads comunes.

Ver <http://java.sun.com/javase/6/docs/api/index.html>

En esta sección se recogen de forma no exhaustiva algunas de las clases de uso más frecuente.

1. **AtomicBoolean (class) java.util.concurrent.atomic**

```
boolean compareAndSet(boolean expect, boolean update)
```

Compara el valor actual con “expect” y, si son iguales (equals), carga el valor “update”.

```
boolean get()
```

Getter.

```
boolean getAndSet(boolean newValue)
```

Carga “newValue” y devuelve el valor anterior.

```
void lazySet(boolean newValue)
```

Sin prisas, cuando pueda, carga el valor “newValue”.

```
void set(boolean newValue)
```

Setter.

```
String toString()
```

Representación para imprimir.

```
boolean weakCompareAndSet(boolean expect, boolean update)
```

Compara el valor actual con “expect” y, si son iguales (==), carga el valor “update”.

2. **AtomicInteger (class) java.util.concurrent.atomic**

Un valor entero sobre el que se definen unos cuantos métodos con la garantía de que están synchronized.

Algunos métodos de interés:

```
AtomicInteger()
```

```
AtomicInteger(int init)
```

```
int get()
```

```
return n;
```

```
void set(int update)
```

```
n = update;
```

```
int getAndSet(int update)
```

```
int x= n; n = update; return x;
```

```
int incrementAndGet()
```

```
n++; return n;
```

```
int addAndGet(int delta)
```

```

        n += delta; return n;

int decrementAndGet()
    n--; return n;

boolean compareAndSet(int expect, int update)
    if (n == expect) { n = update; return true; } else return false;

```

3. ***AtomicIntegerArray (class) java.util.concurrent.atomic***

Un array donde cada elemento se comporta como un *AtomicInteger*.

Algunos métodos de interés:

```

AtomicIntegerArray(int length)
    AtomicInteger[] array = new AtomicInteger[length];

AtomicInteger(int[] init)
    AtomicInteger[] array = new AtomicInteger[init.length];
    for (int i= 0; i < init.length; i++)
        array[i].set(init[i]);

int get(i)
    return array[i].get();

void set(int I, int update)
    return array[i].set(update);

int getAndSet(int i, int update)
    return array[i].getAndSet(update);

int incrementAndGet(int i)
    return array[i].incrementAndGet();

int addAndGet(int I, int delta)
    return array[i].addAndGet(delta);

int decrementAndGet(int i)
    return array[i].decrementAndGet();

boolean compareAndSet(int i, int expect, int update)
    return array[i].compareAndSet(expect, update);

```

4. **AtomicLong (class) java.util.concurrent.atomic**

Un valor long sobre el que se definen unos cuantos métodos con la garantía de que están synchronized.

Ver "[AtomicInteger](#)".

5. **AtomicLongArray (class) java.util.concurrent.atomic**

Un array donde cada elemento se comporta como un [AtomicLong](#).

Ver "[AtomicIntegerArray](#)".

6. **BlockingQueue<E> (interface) java.util.concurrent**

```
public interface BlockingQueue<E>
    extends Queue<E>
```

Hay varias clases que implementan esta interface:

```
public class ArrayBlockingQueue<E>
    extends AbstractQueue<E>
    implements BlockingQueue<E>

public class LinkedBlockingQueue<E>
    extends AbstractQueue<E>
    implements BlockingQueue<E>

public class PriorityBlockingQueue<E>
    extends AbstractQueue<E>
    implements BlockingQueue<E>
```

Los métodos más interesantes son los de meter en la cola y sacar:

```
void put(E e)
```

Añade el elemento a la cola, si es posible. Si no es posible, se queda esperando hasta que puede.

```
E take()
```

Saca un elemento de la cola, si puede. Si no es posible, se queda esperando hasta que puede.

```
int remainingCapacity()
```

Devuelve cuántos elementos caben en la cola en este momento (posiciones libres para meter). Si no hay sitio, devuelve 0. Si la cola no tiene límite, devuelve Integer.MAX_VALUE.

Ver ejemplo en "[productores-consumidores](#)".

6.1. Ejemplo

En este ejemplo, una tarea genera números de la serie de Fibonacci y se los pasa a otra para que los imprima.

class Fibonacci
<pre>public class Fibonacci { public static void main(String[] args) { BlockingQueue<Integer> cola=</pre>

```

        new LinkedBlockingDeque<Integer>();
        ProduceFibo productor= new ProduceFibo cola);
        ConsumeFibo consumidor= new ConsumeFibo cola);

        productor.start();
        consumidor.start();
    }
}

```

class ProduceFibo

```

public class ProduceFibo
    extends Thread {
    private int f1= 1;
    private int f2= 1;
    private final BlockingQueue<Integer> cola;

    public ProduceFibo(BlockingQueue<Integer> cola) {
        this.cola = cola;
    }

    @Override
    public void run() {
        try {
            while (true) {
                int fn= f1+f2;
                cola.put(fn);
                f1= f2;
                f2= fn;
                sleep(1000);
            }
        } catch (InterruptedException e) {
            // se acaba;
        }
    }
}

```

class ConsumeFibo

```

public class ConsumeFibo
    extends Thread {
    private final BlockingQueue<Integer> cola;

    public ConsumeFibo(BlockingQueue<Integer> cola) {
        this.cola = cola;
    }

    @Override
    public void run() {
        try {
            while (true) {
                int n = cola.take();
                System.out.println(n);
            }
        } catch (InterruptedException e) {
            // se acaba;
        }
    }
}

```

7. **Callable<V> (interface) java.util.concurrent**

```
public interface Callable<V>
```

Ofrece un solo método

```
V call() throws Exception
```

para ser usado con Executors.

Ver ejemplo en "[call\(\)](#)".

8. **ConcurrentMap<K, V> (interface) java.util.concurrent**

```
public interface ConcurrentMap<K, V>  
    extends Map<K, V>
```

```
public class ConcurrentHashMap<K, V>  
    extends AbstractMap<K, V>  
    implements ConcurrentMap<K, V>
```

Por una parte, los métodos de la clase Map<K, V> se bloquean para que sean atómicos. Por otra, se añaden algunos métodos nuevos, todos atómicos:

```
V putIfAbsent(K key, V value)
```

```
boolean remove(Object key, Object value)
```

```
V replace(K key, V value)
```

```
boolean replace(K key, V oldValue, V newValue)
```

9. **Condition (interface) java.util.concurrent.locks**

```
public interface Condition
```

Objetos que se usan en combinación con los Locks. Sirven para retener threads esperando hasta que se da una condición que los habilita para proseguir. Un Lock puede tener varias Condition asociadas y de esta forma cada thread espera a su condición concreta.

Ver "[Lock](#)".

```
Condition lock.newCondition()
```

Fábrica (factory) de objetos Condition asociados a un objeto Lock.

```
void await()
```

El thread llamante se mete en la cola hasta que llegue una signal() que lo reactive.

```
void signal()
```

Reactiva al primer thread de la cola.

```
void signalAll()
```

Reactiva a todos los threads de la cola.

Ver ejemplo en "[productores-consumidores](#)".

Ver ejemplo en "[condición](#)".

10. **CopyOnWriteArrayList<E> (class) java.util.concurrent**

```
public class CopyOnWriteArrayList<E>  
    implements List<E>
```

Básicamente, implementa los métodos de la interfaz de forma que sean atómicos. Y, para que no haya interferencias mientras se recorre la lista, los iteradores se hacen con una copia de la lista al principio, de forma que su copia no cambia durante el recorrido.

10.1. Ejemplo

Tenemos un generador que va cargando una lista compartida, añadiendo 1 carácter cada segundo. Concurrentemente, el thread principal recorre el array copiando su contenido en una lista privada. Nótese cómo las adiciones del generador no se reflejan en la iteración en curso, pero sí en la siguiente iteración:

```
class Generador
import java.util.List;
import java.util.TimerTask;

public class Generador extends TimerTask {
    private final List<Character> compartida;
    private char c;

    public Generador(List<Character> compartida) {
        this.compartida = compartida;
        c = 'A';
    }

    @Override
    public void run() {
        System.out.println("Generador: mete: " + c);
        compartida.add(c);
        c++;
    }
}

class ConcurrentList
import java.util.ArrayList;
import java.util.List;
import java.util.Timer;
import java.util.concurrent.CopyOnWriteArrayList;

public class ConcurrentList {
    public static void main(String[] args)
        throws InterruptedException {
        List<Character> compartida = new CopyOnWriteArrayList<Character>();

        Generador generador = new Generador(compartida);
        Timer timer = new Timer();
        timer.schedule(generador, 1000, 1000);

        while (true) {
            System.out.println("start");
            List<Character> miLista = new ArrayList<Character>();
            for (Character ch : compartida) {
                miLista.add(ch);
                Thread.sleep(1000);
            }
            System.out.println("end: " + miLista);
            System.out.println();
            Thread.sleep(1500);
        }
    }
}
```

```

    }
}

> java ConcurrentList

start
end: []

Generador: mete: A
start
Generador: mete: B
end: [A]

Generador: mete: C
Generador: mete: D
start
Generador: mete: E
Generador: mete: F
Generador: mete: G
Generador: mete: H
end: [A, B, C, D]

Generador: mete: I
start
Generador: mete: J
Generador: mete: K
Generador: mete: L
Generador: mete: M
Generador: mete: N
Generador: mete: O
Generador: mete: P
Generador: mete: Q
Generador: mete: R
end: [A, B, C, D, E, F, G, H, I]
...

```

11. CopyOnWriteArraySet<E> (class) java.util.concurrent

```
public class CopyOnWriteArraySet<E>
implements Set<E>
```

Básicamente, implementa los métodos de la interfaz de forma que sean atómicos. Y, para que no haya interferencias mientras se recorre el conjunto, los iteradores se hacen con una copia al principio, de forma que su copia no cambia durante el recorrido.

Ver "[CopyOnWriteArrayList](#)".

12. CountdownLatch (class) java.util.concurrent

Clase sincronizadora. Se establece una cuenta inicial. Las clases se apuntan al contador, que se va decrementando y, cuando llega a cero, todas las clases que esperan se reactivan.

12.1. Ejemplo 1. Señal de salida: arranque sincronizado

```

class Tarea1

import java.util.concurrent.CountDownLatch;

public class Tarea1 extends Thread {

```

```

private CountdownLatch signal;

public Tareal(CountDownLatch signal) {
    this.signal = signal;
}

public void run() {
    try {
        signal.await();
    } catch (InterruptedException ignore) {
    }
    System.out.println("empiezo ...");
    // hace su tarea
}
}

```

class SalidaSincronizada

```

import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.CountDownLatch;

public class SalidaSincronizada {
    public static void main(String[] args)
        throws InterruptedException {
        CountdownLatch signal = new CountdownLatch(0);
        List<Tareal> tareas = new ArrayList<Tareal>();
        tareas.add(new Tareal(signal));
        tareas.add(new Tareal(signal));
        tareas.add(new Tareal(signal));
        // ... N veces

        for (Tareal tarea : tareas)
            tarea.start();

        System.out.println("ya");
        // todos empiezan a la señal
        signal.countDown();
    }
}

```

12.2. Ejemplo 2. Señal de fin: sincronización de llegada

class Tarea2

```

import java.util.concurrent.CountDownLatch;

public class Tarea2 extends Thread {
    private CountdownLatch signal;

    public Tarea2(CountDownLatch signal) {
        this.signal = signal;
    }

    public void run() {
        // hace su tarea
        System.out.println("termino ...");
        signal.countDown();
    }
}

```

class LlegadaSincronizada

```
import java.util.concurrent.CountDownLatch;

public class LlegadaSincronizada {
    private static final int N = 10;

    public static void main(String[] args)
        throws InterruptedException {
        Tarea2[] tareas = new Tarea2[N];
        CountDownLatch signal =
            new CountDownLatch(tareas.length);
        for (int i = 0; i < tareas.length; i++)
            tareas[i] = new Tarea2(signal);

        for (Tarea2 tarea : tareas)
            tarea.start();

        // espero N señales
        signal.await();
        System.out.println("fin");
    }
}
```

13. CyclicBarrier (class) java.util.concurrent

Clase sincronizadora. Se establece una cuenta inicial de participantes. Las clases se apuntan al contador, que se va decrementando y, cuando llega a cero, todas las clases que esperan se reactivan y el contador se resetea a su valor inicial.

CyclicBarrier(int parties)
Constructor.

CyclicBarrier(int parties, Runnable action)
Constructor. Cada vez que todas las partes se sincronizan, se ejecuta la acción asociada.

int await()
Espero a que estemos todas las partes.

13.1. Ejemplo

Tres amigos se sincronizan regularmente. Además tenemos una escucha global que registra la hora del encuentro.

class Amigo

```
import java.util.concurrent.CyclicBarrier;

public class Amigo extends Thread {
    private final String name;
    private final CyclicBarrier barrier;

    public Amigo(String name, CyclicBarrier barrier) {
        this.name = name;
        this.barrier = barrier;
    }

    public void run() {
        while (true) {
            try {
```

```

        sleep((int) (10000 * Math.random()));
        System.out.println(name + " esperando ...");
        barrier.await();
    } catch (Exception e) {
        e.printStackTrace();
    }
}
}
}

```

class Amigos

```

import java.util.Date;
import java.util.concurrent.CyclicBarrier;

public class Amigos {
    public static void main(String[] args) {
        Escucha escucha = new Escucha();
        CyclicBarrier cita = new CyclicBarrier(3, escucha);
        Amigo pedro = new Amigo("Pedro", cita);
        Amigo angel = new Amigo("Angel", cita);
        Amigo bruno = new Amigo("Bruno", cita);

        pedro.start();
        angel.start();
        bruno.start();
    }

    private static class Escucha implements Runnable {
        public void run() {
            System.out.println(new Date());
        }
    }
}

```

14. Exchanger (class) java.util.concurrent

Clase sincronizadora. Define un punto de encuentro en el que dos threads se encuentran e intercambian un objeto entre ellos.

Exchanger<V>()

Constructor. Intercambia objetos de clase V.

V exchange(V x)

Yo envíé x, y recibo otro objeto del mismo tipo.

14.1. Ejemplo

class Amigo

```

import java.util.concurrent.CountDownLatch;

public class Tarea2 extends Thread {
    private CountDownLatch signal;

    public Tarea2(CountDownLatch signal) {
        this.signal = signal;
    }

    public void run() {

```

<pre> // hace su tarea System.out.println("termino ..."); signal.countDown(); } } </pre>
<p>class Amigos</p> <pre> import java.util.concurrent.Exchanger; public class Amigo extends Thread { private final String name; private final Exchanger<String> cita; public Amigo(String name, Exchanger<String> cita) { this.name = name; this.cita = cita; } public void run() { try { sleep((int) (10 * Math.random())); System.out.println(name + " esperando ..."); String recibido = cita.exchange("regalo de " + name); System.out.println(name + ": " + recibido); } catch (InterruptedException ignore) { } } } </pre>
<p>> java Amigos</p> <pre> Angel esperando ... Pedro esperando ... Pedro: regalo de Angel Angel: regalo de Pedro </pre>

15. **Executor (interface) java.util.concurrent**

```
public interface Executor
```

Para controlar la ejecución de threads.

Ver ejemplo en "[ejecutor](#)".

16. **Executors (factory) java.util.concurrent**

```
public class Executors
```

Proporciona fábricas (factories) para generar diferentes tipos de Executor Services.

Se recogen algunos de las fábricas más usadas:

```
static ExecutorService newCachedThreadPool()
```

Se crean nuevos Threads según se necesitan; pero si un thread termina, se recicla antes que crear uno nuevo.

```
static ExecutorService newFixedThreadPool(int maximo)
```

Se utiliza hasta un máximo número de Threads. Si hay más solicitudes que Threads, esperan a que termine alguno.

```
static ExecutorService newSingleThreadExecutor()
```

Utiliza un (1) Thread. Si hay varias solicitudes, esperan a que termine.

```
static ScheduledExecutorService newScheduledThreadPool(int maximo)
```

Se utiliza hasta un máximo número de Threads. Si hay más solicitudes que Threads, esperan a que termine alguno.

```
static ScheduledExecutorService newSingleThreadScheduledExecutor()
```

Utiliza un (1) Thread. Si hay varias solicitudes, esperan a que termine.

Ver ejemplo en "[ejecutor](#)".

17. **ExecutorService (interface) java.util.concurrent**

```
public interface ExecutorService
    extends Executor
```

Son gestores de Threads. Eso quiere decir que disponen de varios objetos de tipo Thread que van utilizando "sabiamente" para ejecutar las diferentes tareas que se le pasen para ejecutar.

```
Future<?> submit(Runnable task)
```

Para que se ejecute un Runnable.

```
<T> Future<T> submit(Callable<T> task)
```

Para que se ejecute un Callable<T>.

```
void shutdown()
```

Terminación ordenada. No se aceptan nuevos threads, pero se espera a que terminen los que están pendientes.

```
List<Runnable> shutdownNow()
```

Terminación forzada. Intenta parar los threads en ejecución, y devuelve los que quedan pendientes.

Ver ejemplo en "[ejecutor](#)".

18. **Future<V> (interface)**

```
public interface Future<V>
```

Son objetos utilizados por los Executors para devolver el resultado de la ejecución de un thread Callable<V>.

Sea por ejemplo una clase que devuelve una String

```
class Worker implements Callable<String> {
    public String call() throws Exception {
        ...
        return string;
    }
}
```

Creamos un objeto de dicha clase

```
Callable<String> worker = new T();
```

Si tenemos un ExecutorService

```
ExecutorService service = ...
```

le podemos pasar el objeto para que lo lance como thread

```
Future<String> future = service.submit(worker);
```

Este future de momento es mera promesa.

Cuando queramos saber el valor calculado

```
String valor = future.get();
```

La llamada a get() bloquea la ejecución hasta que el thread worker ha completado su cálculo.

Ver ejemplo en "[call\(\)](#)".

V get()

Espera hasta que el thread termine y devuelve el resultado.

V get(long time, TimeUnit unit)

Espera hasta un cierto tiempo máximo. Si está el valor disponible, lo devuelve. Si no, lanza una interrupción.

boolean isDone()

Devuelve TRUE si el thread ha terminado y el valor está disponible.

19. Lock (interface) java.util.concurrent.locks

```
public class ReentrantLock  
    implements Lock
```

Es otro mecanismo para crear zonas de exclusión mutua.

Los Locks parecen redundantes. Java los proporciona porque además se pueden asociar condiciones, generalizando el papel de los métodos wait() y notify().

void lock()

Adquiere el cerrojo. Si es propiedad de otro thread espera a que se libere. Si es propiedad del mismo thread que lo solicita, se concede inmediatamente (y se lleva cuenta de las solicitudes, que deberán ser descontadas, unlock(), una a una).

void lockInterruptibly() throws InterruptedException

Como lock() pero es posible interrumpir la espera.

void unlock()

Libera el cerrojo.

Condition newCondition()

Crea una condición de espera (o sea, una cola de espera) para tareas que desean adquirir el cerrojo.

Ver ejemplo en "[productores-consumidores](#)".

Ver ejemplo en "[condición](#)".

20. *PipedInputStream* (class) java.io

```
public class PipedInputStream
    extends InputStream
```

Junto con *PipedOutputStream*, permite crear canales asíncronos para comunicar threads.

```
PipedInputStream()
```

Crea el objeto sin conectarlo a un output stream.

```
PipedInputStream (int pipeSize)
```

Crea el objeto sin conectarlo a un output stream. Impone un tamaño máximo de caracteres en tránsito.

```
PipedInputStream (PipedOutputStream output)
```

Crea el objeto y lo conecta a un output stream.

```
PipedInputStream (PipedOutputStream output, int pipeSize)
```

Crea el objeto y lo conecta a un output stream. Impone un tamaño máximo de caracteres en tránsito.

```
void connect(PipedOutputStream output)
```

Conecta el objeto con un output stream.

Ver ejemplo en "[*productores-consumidores*](#)".

21. *PipedOutputStream* (class) java.io

```
public class PipedOutputStream
    extends OutputStream
```

Junto con *PipedInputStream*, permite crear canales asíncronos para comunicar threads.

```
PipedOutputStream()
```

Crea el objeto sin conectarlo a un input stream.

```
PipedOutputStream (PipedInputStream input)
```

Crea el objeto y lo conecta a un input stream.

```
void connect(PipedInputStream input)
```

Conecta el objeto con un input stream.

Ver ejemplo en "[*productores-consumidores*](#)".

22. *PipedReader* (class) java.io

```
public class PipedReader
    extends Reader
```

Similar a *PipedInputStream*, pero trabaja sobre caracteres.

```
PipedReader()
```

Crea el objeto sin conectarlo a un writer.

```
PipedReader(int pipeSize)
```

Crea el objeto sin conectarlo a un writer. Impone un tamaño máximo de caracteres en tránsito.

```
PipedReader(PipedWriter writer)
```

Crea el objeto y lo conecta a un writer.

```
PipedReader(PipedWriter src, int pipeSize)
```

Crea el objeto y lo conecta a un writer. Impone un tamaño máximo de caracteres en tránsito.

```
void connect(PipedWriter writer)
```

Conecta el objeto con un writer.

23. PipedWriter (class) java.io

```
public class PipedWriter  
    extends Writer
```

Similar a *PipedOutputStream*, pero trabaja sobre caracteres.

```
PipedWriter()
```

Crea el objeto sin conectarlo a un reader.

```
PipedWriter(PipedReader reader)
```

Crea el objeto y lo conecta a un reader.

```
void connect(PipedReader reader)
```

Conecta el objeto con un reader.

24. ReadWriteLock (interface) java.util.concurrent.locks

```
public interface ReadWriteLock
```

Las clases que implementan esta interfaz, como *ReentrantReadWriteLock*, permiten obtener dos cerrojos sincronizados:

```
ReentrantReadWriteLock rwl = new ReentrantReadWriteLock();  
Lock readLock = rwl.readLock();  
Lock writeLock = rwl.writeLock();
```

De forma que los cerrojos *readLock* y *writeLock* ya llevan la cuenta de que pueden haber varios readers a la vez, pero sólo un writer, y que hay que darle prioridad al writer:

Ver ejemplo en "*readers & writers*".

25. Runnable (interface) java.lang

```
public interface Runnable
```

Ofrece un solo método

```
void run()
```

para ser usado en *threado conejecutor*.

26. ScheduledExecutorService (interface) java.util.concurrent

```
public interface ScheduledExecutorService  
    extends ExecutorService
```

Variante de *Executor* que no ejecuta el thread inmediatamente, sino dentro de un tiempo.

```
<V> ScheduledFuture<V> schedule(Callable<V> task,  
                                long delay, TimeUnit unit)
```

```
ScheduledFuture<V> schedule(Runnable task,  
                              long delay, TimeUnit unit)
```

```
ScheduledFuture<V> scheduleAtFixedRate(Runnable task,  
                                         long initial, long period, TimeUnit unit)
```

```
ScheduledFuture<V> scheduleAtFixedDate(Runnable task,  
                                         long initial, long delay, TimeUnit unit)
```

27. *ScheduledFuture<V>* (interface) *java.util.concurrent*

```
public interface ScheduledFuture<V>  
    extends Delayed, Future<V>
```

Baste decir que es un refinamiento de *Future* para incorporar un retraso en su evaluación.

```
V get()
```

Espera hasta que el thread termine y devuelve el resultado.

28. *Semaphore* (class) *java.util.concurrent*

Ver "*semáforo*"

```
Semaphore(int permits)
```

Crea un semáforo con un cierto número de permisos.

```
public void acquire(int permits) throws InterruptedException  
    Solicita unos permisos. Si no hay suficientes, queda esperando. Si los hay, se descuentan  
    del semáforo y se sigue. Si llega una interrupción, se aborta la espera.
```

```
public void acquire() throws InterruptedException  
    acquire(1)
```

```
public void acquireUninterruptibly(int permits)  
    Solicita unos permisos. Si no hay suficientes, queda esperando. Si los hay, se descuentan  
    del semáforo y se sigue. Si llega una interrupción, se ignora y se sigue esperando.
```

```
public void acquireUninterruptibly()  
    acquireUninterruptibly(1)
```

```
release(int permits)
```

Libera unos permisos que retornan a la cuenta del semáforo. Si hay threads esperando, se reactivan.

```
release()  
    release(1)
```

29. *Thread* (clase) *java.lang*

```
public class Thread  
    implements Runnable
```

Los métodos más relevantes se han despiezado en la primera parte de este vademécum.

Estado

enum Thread.State

NEW

El thread se ha creado, pero aún no ha arrancado.

RUNNABLE

El thread está funcionando.

BLOCKED

El thread está esperando en una zona de exclusión mutua.

WAITING

El thread está esperando a otro thread. No hay un tiempo máximo de espera, el thread espera indefinidamente.

— wait()

— join()

TIMED_WAITING

El thread está esperando a otro thread, con un tiempo máximo de espera.

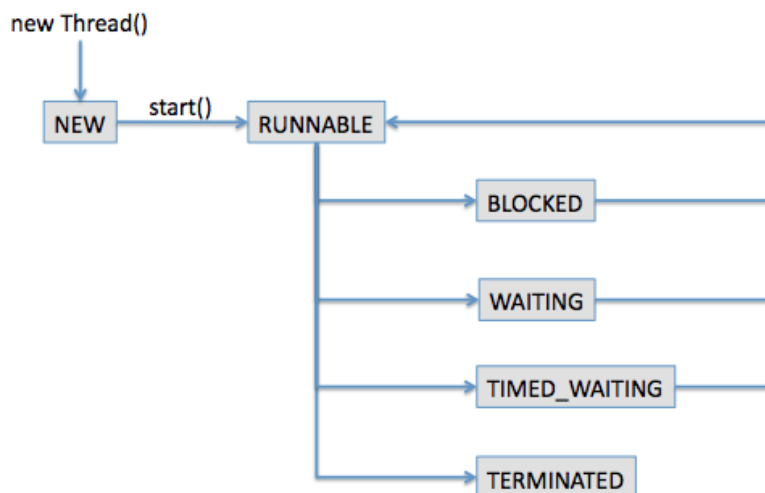
— Thread.sleep(t)

— wait(t)

— join(t)

TERMINATED

El método run() ha llegado al final.



Prioridad

Cada thread puede tener una prioridad. La prioridad de un thread es un entero. En líneas generales, puede decirse que si dos threads están listos para ejecutarse, se ejecutará antes el de mayor prioridad.

Se definen algunas constantes, obvias:

Thread.MAX_PRIORITY

```
Thread.NORM_PRIORITY
Thread.MIN_PRIORITY
```

30. ThreadGroup (class) java.lang

Una forma de agrupar varios threads bajo un nombre común, a efectos de estructurar un programa que maneja muchos threads, permitiendo algunas acciones sobre el conjunto de threads.

31. Timer (class) java.util

```
public class Timer
```

Permite programar una serie de threads para que se ejecuten al cabo de cierto tiempo, o cada cierto tiempo.

```
Timer()
```

Constructor.

```
Timer(boolean isDaemon)
```

Constructor. Puede funcionar como demonio.

```
Timer(String name)
```

Constructor con nombre propio.

```
Timer(String name, boolean isDaemon)
```

Constructor con nombre propio. Puede funcionar como demonio.

```
void schedule(TimerTask task, Date date)
```

La tarea se ejecutará en la fecha indicada.

```
void schedule(TimerTask task, Date date, long period)
```

La tarea se ejecutará en la fecha indicada y a partir de entonces se repetirá periódicamente.

```
void schedule(TimerTask task, long delay)
```

La tarea se ejecutará dentro de un tiempo.

```
void schedule(TimerTask task, long delay, long period)
```

La tarea se ejecutará dentro de un tiempo y a partir de entonces se repetirá periódicamente.

```
void scheduleAtFixedRate(TimerTask task, Date first, long period)
```

La tarea se ejecutará en la fecha indicada y a partir de entonces se repetirá periódicamente regularmente cada cierto tiempo.

```
void scheduleAtFixedRate(TimerTask task, long delay, long period)
```

La tarea se ejecutará dentro de un tiempo y a partir de entonces se repetirá periódicamente regularmente cada cierto tiempo.

31.1. Ejemplo

Una tarea que se ejecuta cada minuto.

```
class Reloj
```

```
import java.util.Date;
import java.util.TimerTask;
```

```
public class Reloj extends TimerTask {
```

```
@Override
public void run() {
    System.out.println(new Date());
}
}
```

class Planificador

```
import java.util.Date;
import java.util.Timer;
import java.util.TimerTask;

public class Planificador {
    public static void main(String[] args) {
        Timer timer = new Timer();
        TimerTask task = new Reloj();

        System.out.println(new Date());
        timer.schedule(task, 10000, 30000);
    }
}
```

> java Planificador

```
Fri Jan 06 14:04:51 CET 2012
Fri Jan 06 14:05:01 CET 2012
Fri Jan 06 14:05:31 CET 2012
Fri Jan 06 14:06:01 CET 2012
Fri Jan 06 14:06:31 CET 2012
Fri Jan 06 14:07:01 CET 2012
...
```

32. *TimerTask* (class) *java.util*

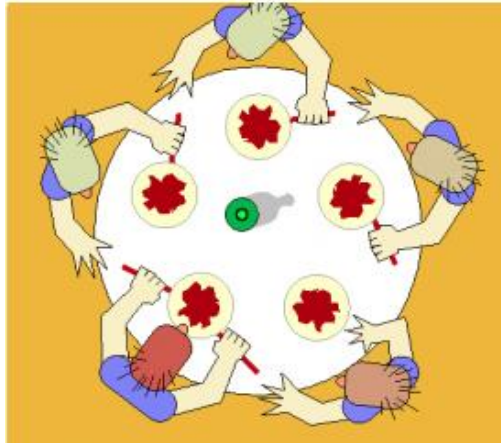
```
public abstract class TimerTask
implements Runnable
```

Básicamente un envoltorio de clases Runnable para pasarlo a un *Timer*.

Problemas clásicos

1. Dining philosophers (problema)

Problema clásico de concurrencia. 5 filósofos se disponen alrededor de una mesa. Hay 5 tenedores para compartir.



[imagen tomada de <http://www.xjtek.com/>]

Cuando un filósofo consigue sus 2 tenedores, puede comer. Hay un riesgo de bloqueo (deadlock) si los 5 filósofos cogen su tenedor izquierdo y quedan esperando a poder coger su tenedor derecho.

El objeto es analizar cómo varios threads pueden adquirir derechos sobre varios recursos compartidos, sin bloquearse mutuamente.

Solución 1: monitor

Una solución sencilla es usar un monitor para controlar el acceso atómico a los dos tenedores que necesita un filósofo para comer:

class DiningPhilosophers

```
public class DiningPhilosophers {
    private static final int N = 5;

    public static void main(String[] args) {
        Table table = new Table(N);
        for (int i = 0; i < N; i++) {
            Philosopher philosopher =
                new Philosopher(table, i);
            philosopher.start();
        }
    }
}
```

enum Philosopher

```
public class Philosopher extends Thread {
    private final int id;
    private final Table table;

    public Philosopher(Table table, int id) {
        this.table = table;
        this.id = id;
    }
}
```

```

private void setState(String state) {
    System.out.println(id + ": " + state);
    try {
        sleep(2 * 1000);
    } catch (InterruptedException ignored) {
    }
}

public void run() {
    try {
        while (true) {
            setState("THINKING");
            table.aComer(id);
            setState("EATING");
            table.aPensar(id);
        }
    } catch (InterruptedException ignore) {
    }
}
}

```

class Table (monitor)

```

public class Table {
    // true si el tenedor está en uso
    private final boolean[] fork;

    public Table(int n) {
        fork = new boolean[n];
    }

    private int left(int i) {
        return i == 0 ? fork.length - 1 : i - 1;
    }

    private int right(int i) {
        return i;
    }

    public synchronized void aComer(int philo)
        throws InterruptedException {
        while (fork[left(philos)] || fork[right(philos)])
            wait();
        fork[left(philos)] = true;
        fork[right(philos)] = true;
    }

    public synchronized void aPensar(int philo) {
        fork[left(philos)] = false;
        fork[right(philos)] = false;
        notifyAll();
    }
}

```

> java DiningPhilosophers

```

0: THINKING
1: THINKING
2: THINKING
3: THINKING
4: THINKING

```

```
0: EATING
2: EATING
0: THINKING
4: EATING
2: THINKING
1: EATING
4: THINKING
1: THINKING
3: EATING
...
```

Solución 2: semáforo

La solución que se muestra a continuación incluye zonas synchronized alrededor del objeto TABLE para que ciertas partes del código se ejecuten como críticas, y un semáforo para que un filósofo quede listo para comer y coma efectivamente.

Quizás se puede acusar a esta solución de excesivamente artesanal.

class DiningPhilosophers

```
public class DiningPhilosophers {
    public static void main(String[] args) {
        Object table = new Object();
        Philosopher ph0 = new Philosopher(table, 0);
        Philosopher ph1 = new Philosopher(table, 1);
        Philosopher ph2 = new Philosopher(table, 2);
        Philosopher ph3 = new Philosopher(table, 3);
        Philosopher ph4 = new Philosopher(table, 4);

        ph0.setLeft(ph4); ph0.setRight(ph1);
        ph1.setLeft(ph0); ph1.setRight(ph2);
        ph2.setLeft(ph1); ph2.setRight(ph3);
        ph3.setLeft(ph2); ph3.setRight(ph4);
        ph4.setLeft(ph3); ph4.setRight(ph0);

        ph0.start();
        ph1.start();
        ph2.start();
        ph3.start();
        ph4.start();
    }
}
```

enum PhState

```
public enum PhState {
    THINKING, HUNGRY, EATING
}
```

class Philosopher

```
import java.util.concurrent.Semaphore;

public class Philosopher extends Thread {
    private PhState state;
    private final int id;
    private Philosopher left, right;

    private final Object table;
    private final Semaphore semaphore;
```

```

public Philosopher(Object table, int id) {
    this.table = table;
    this.id = id;
    semaphore = new Semaphore(1);
    try {
        semaphore.acquire();
    } catch (InterruptedException ignore) {
    }
}

public void setLeft(Philosopher philo) {
    left = philo;
}

public void setRight(Philosopher philo) {
    right = philo;
}

private void setState(PhState newState) {
    state = newState;
    System.out.println(id + ": " + state);
    try {
        sleep(2 * 1000);
    } catch (InterruptedException ignore) {
    }
}

public void run() {
    while (true) {
        setState(PhState.THINKING);
        takeForks();
        putForks();
    }
}

public void test() {
    if (state == PhState.HUNGRY
        && left.state != PhState.EATING
        && right.state != PhState.EATING) {
        setState(PhState.EATING);
        semaphore.release();
    }
}

private void takeForks() {
    synchronized (table) {
        setState(PhState.HUNGRY);
        test();
    }
    try {
        semaphore.acquire();
    } catch (InterruptedException ignore) {
    }
}

private void putForks() {
    synchronized (table) {
        setState(PhState.THINKING);
        left.test();
        right.test();
    }
}

```


throws InterruptedException;
}
class Productor
<pre>import java.util.Random; public class Productor extends Thread { private final Random random = new Random(); private final Buffer<Character> buffer; private final char c; public Productor(Buffer<Character> buffer, char c) { this.buffer = buffer; this.c = c; } public void run() { for (int i = 0; i < 10; i++) { try { buffer.put((char) (c + i)); Thread.sleep(random.nextInt(5) * 1000); } catch (InterruptedException ignore) { } } } }</pre>
class Consumidor
<pre>import java.util.Random; public class Consumidor extends Thread { private final Random random = new Random(); private final Buffer<Character> buffer; public Consumidor(Buffer<Character> buffer) { this.buffer = buffer; } public void run() { while (true) { try { Character msg = buffer.get(); Thread.sleep(random.nextInt(2) * 1000); } catch (InterruptedException ignore) { } } } }</pre>

Solución 1: wait & notify

La clase Buffer se encarga de parar y arrancar al productor según la disponibilidad de espacio para retener datos y al consumidor según haya datos disponibles:

class Buffer1<E> implements Buffer<E>
<pre>public class Buffer1<E> implements Buffer<E> {</pre>

```

private E[] data;
private int nDatos;

public Buffer1(int size) {
    data = (E[]) new Object[size];
    nDatos = 0;
}

public synchronized void put(E x)
    throws InterruptedException {
    while (nDatos >= data.length)
        wait();
    data[nDatos++] = x;
    notifyAll();
}

public synchronized E get()
    throws InterruptedException {
    while (nDatos <= 0)
        wait();
    nDatos--;
    E x = data[0];
    System.arraycopy(data, 1, data, 0, nDatos);
    data[nDatos] = null;
    notifyAll();
    return x;
}
}

```

Solución 2: con semáforos

class Buffer2<E> implements Buffer<E>

```

import java.util.concurrent.Semaphore;

public class Buffer2<E>
    implements Buffer<E> {
    private final E[] data;
    private int nDatos;

    private final Semaphore haySitio;
    private final Semaphore hayDatos;

    public Buffer2(int size) {
        data = (E[]) new Object[size];
        nDatos = 0;
        haySitio = new Semaphore(size);
        hayDatos = new Semaphore(0);
    }

    public void put(E x)
        throws InterruptedException {
        haySitio.acquire();
        try {
            synchronized (data) {
                data[nDatos++] = x;
                print();
            }
        }
    }
}

```

```

        } finally {
            hayDatos.release();
        }
    }

    public E get()
        throws InterruptedException {
        hayDatos.acquire();
        try {
            E x;
            synchronized (data) {
                nDatos--;
                x = data[0];
                System.arraycopy(data, 1, data, 0, nDatos);
                data[nDatos] = null;
                print();
            }
            return x;
        } finally {
            haySitio.release();
        }
    }
}

```

Solución 3: locks & conditions

Variante del uso de wait & notify. Aquí se organizan diferentes colas de espera alrededor de diferentes condiciones asociadas a un cierto lock.

Revisamos la clase Buffer se encarga de parar y arrancar al productor según la disponibilidad de espacio para retener datos y al consumidor según haya datos disponibles:

class Buffer3<E> implements Buffer<E>

```

import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

public class Buffer3<E>
    implements Buffer<E> {
    private final E[] data;
    private int nDatos;

    private final Lock llave;
    private final Condition haySitio;
    private final Condition hayDatos;

    public Buffer3(int size) {
        data = (E[]) new Object[size];
        nDatos = 0;
        llave = new ReentrantLock();
        hayDatos = llave.newCondition();
        haySitio = llave.newCondition();
    }

    public void put(E x)
        throws InterruptedException {
        llave.lock();
        try {
            while (nDatos >= data.length)

```

```

        haySitio.await();
        data[nDatos++] = x;
        hayDatos.signal();
    } finally {
        llave.unlock();
    }
}

public E get()
    throws InterruptedException {
    llave.lock();
    try {
        while (nDatos <= 0)
            hayDatos.await();
        nDatos--;
        E x = data[0];
        System.arraycopy(data, 1, data, 0, nDatos);
        data[nDatos] = null;
        haySitio.signal();
        return x;
    } finally {
        llave.unlock();
    }
}
}

```

Solución 4: BlockingQueue

class Buffer4<E> implements Buffer<E>

```

import java.util.concurrent.BlockingQueue;
import java.util.concurrent.LinkedBlockingQueue;

public class Buffer4<E>
    implements Buffer<E> {
    private final BlockingQueue<E> queue;

    public Buffer4(int size) {
        queue = new LinkedBlockingQueue<E>();
    }

    public void put(E x)
        throws InterruptedException {
        queue.put(x);
    }

    public E get()
        throws InterruptedException {
        return queue.take();
    }
}

```

Ver "[BlockingQueue](#)".

Solución 5: Piped streams

El autor debe hacer constar que esta forma de comunicar threads en java parece frágil en el sentido de que cualquier desviación del patrón lanza múltiples excepciones y pueden dejar las colas inservibles.

class Buffer5<E> implements Buffer<E>

```
import java.io.*;

public class Buffer5<E>
    implements Buffer<E> {
    private PipedOutputStream pos;
    private PipedInputStream pis;
    private ObjectOutputStream oos;
    private ObjectInputStream ois;

    public Buffer5(int size) {
        try {
            PipedOutputStream pos = new PipedOutputStream();
            PipedInputStream pis = new PipedInputStream(pos);
        } catch (IOException e) {
            System.err.println("new Buffer: " + e);
        }
    }

    public void put(E x) {
        try {
            if (oos == null)
                oos = new ObjectOutputStream(pos);
            oos.writeObject(x);
        } catch (IOException e) {
            System.err.println("Buffer.put " + e);
        }
    }

    public E get() {
        try {
            if (ois == null)
                ois = new ObjectInputStream(pis);
            return (E) ois.readObject();
        } catch (Exception ioe) {
            System.err.println("Buffer.get " + ioe);
            return null;
        }
    }
}
```

Ver ["PipedInputStream"](#), ["PipedOutputStream"](#).

Ver ["PipedReader"](#), ["PipedWriter"](#).

3. Readers & writers (problema)

Un caso clásico de estudio. Se trata de un dato compartido al que quieren acceder diferentes threads para leer y diferentes threads para escribir, con la particularidad de que

- varios threads pueden leer a la vez, ya que la lectura no modifica los datos
- sólo uno (1) thread puede estar escribiendo, ya que se modifican los datos

Una solución simple con un monitor puede llevar a que los threads lectores nunca permitan que se ejecuten los escritores, simplemente porque la condición para leer es más frecuente que la condición para escribir.

Una solución más elaborada priorizará los threads escritores deteniendo las operaciones de lectura si hay threads esperando para escribir.

Solución: monitor con condiciones

El código se ha instrumentado con trazas y con unos tiempos que provoquen rápidamente las situaciones conflictivas: escritores esperando.

class Escenario

```
public class Escenario {
    private static final int N_READERS = 4;
    private static final int N_WRITERS = 2;

    public static void main(String[] args) {
        Data0 data = new Data0();
        for (int i = 0; i < N_WRITERS; i++) {
            int id = 10 + 10 * i; // 10, 20, 30, 40, ...
            Writer writer = new Writer(id, data);
            writer.start();
        }
        for (int i = 0; i < N_READERS; i++) {
            Reader reader = new Reader(i, data);
            reader.start();
        }
    }
}
```

class Reader

```
public class Reader extends Thread {
    private final int id;
    private final Data0 data;

    public Reader(int id, Data0 data) {
        this.id = id;
        this.data = data;
    }

    public void run() {
        try {
            while (true) {
                data.openReading();
                int x = data.getValue();
                System.out.println(id + " reads " + x);
                sleep((int) (Math.random() * 2000));
                data.closeReading();
            }
        } catch (InterruptedException ignore) {
        }
    }
}
```

class Writer

```
public class Writer extends Thread {
    private final int id;
    private final Data0 data;
    private int cnt = 100;
```

```

public Writer(int id, Data0 data) {
    this.id = id;
    this.data = data;
}

public void run() {
    try {
        while (true) {
            sleep((int) (Math.random() * 400 * id));
            int x = 10 * id + cnt++;
            data.openWriting();
            sleep((int) (Math.random() * 1000));
            data.setValue(x);
            System.out.println(id + " writes " + x);
            data.closeWriting();
        }
    } catch (InterruptedException ignore) {
    }
}
}

```

class Data0

```

public class Data0 {
    private int x;

    private int nReaders;
    private boolean writing = false;
    private int waitingWriters = 0;

    public int getValue() {
        return x;
    }

    public void setValue(int x) {
        this.x = x;
    }

    public synchronized void openReading()
        throws InterruptedException {
        while (writing || waitingWriters > 0)
            wait();
        nReaders++;
    }

    public synchronized void closeReading()
        throws InterruptedException {
        nReaders--;
        notifyAll();
    }

    public synchronized void openWriting()
        throws InterruptedException {
        while (writing || nReaders > 0) {
            waitingWriters++;
            System.out.printf(
                "openWriting: %d readers; %d waiting writers.%n",
                nReaders, waitingWriters);
            wait();
            waitingWriters--;
        }
    }
}

```

```

        }
        writing = true;
        System.out.println("START WRITING");
    }

    public synchronized void closeWriting()
        throws InterruptedException {
        writing = false;
        System.out.println("END WRITING");
        notifyAll();
    }
}

```

Solución: ReadWriteLock

El problema de los N readers y N writers es tan habitual que la biblioteca java proporciona objetos específicamente dedicados a la tarea. Se trata de dos cerrojos sincronizados entre sí

```

ReadWriteLock rwl = new ReentrantReadWriteLock();
Lock readLock = rwl.readLock();
Lock writeLock = rwl.writeLock();

```

De forma que los cerrojos readLock y writeLock ya llevan la cuenta de que pueden haber varios readers a la vez, pero sólo un writer, y que hay que darle prioridad al writer:

class Data2

```

import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantReadWriteLock;

public class Data2 {
    private int x;

    private final ReentrantReadWriteLock rwl;
    private final Lock readLock;
    private final Lock writeLock;

    public Data2() {
        rwl = new ReentrantReadWriteLock();
        readLock = rwl.readLock();
        writeLock = rwl.writeLock();
    }

    public int getValue() {
        return x;
    }

    public void setValue(int x) {
        this.x = x;
    }

    public void openReading() {
        readLock.lock();
        System.out.println("openReading.getReadLockCount(): " +
            rwl.getReadLockCount());
    }

    public void closeReading() {
        readLock.unlock();
    }
}

```

```

        System.out.println("closeReading.getReadLockCount(): " +
            rwl.getReadLockCount());
    }

    public void openWriting() {
        writeLock.lock();
        writing = true;
        System.out.println("openWriting.getReadLockCount(): " +
            rwl.getReadLockCount());
        System.out.println("START WRITING");
    }

    public void closeWriting() {
        writing = false;
        System.out.println("END WRITING");
        writeLock.unlock();
    }
}

```

> java Escenario

```

openReading.getReadLockCount(): 1
0 reads 0
openReading.getReadLockCount(): 2
1 reads 0
openReading.getReadLockCount(): 3
2 reads 0
openReading.getReadLockCount(): 4
3 reads 0
closeReading.getReadLockCount(): 3
closeReading.getReadLockCount(): 2
closeReading.getReadLockCount(): 1
closeReading.getReadLockCount(): 0
openWriting.getReadLockCount(): 0
START WRITING
20 writes 300
END WRITING
openReading.getReadLockCount(): 1
1 reads 300
openReading.getReadLockCount(): 2
3 reads 300
...

```

Se han añadido algunas trazas para ver cómo se van concediendo accesos para lectura y cómo se prioriza una petición de escritura.

Ver "[ReadWriteLock](#)".

4. Singleton (pattern)

En programación orientada a objetos existe un patrón para los casos en que deseamos que de un cierto objeto haya 1 sola instancia:

```

public class Singleton {
    private static Singleton instancia;

    private Singleton() {
        // lo construye
    }
}

```

```

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

```

Cuando intentamos llevar esto al mundo concurrente, para garantizar que no se creen nunca 2 instancias hay que bloquear la creación:

```

public synchronized Singleton getInstancia() {
    if (instancia == null) {
        instancia = new Singleton();
    }
    return instancia;
}
}

```

Este código es correcto, pero pudiera tener cierto efecto negativo sobre la velocidad de ejecución ya que sincronizar cuesta tiempo y todos los usuarios del getter tienen que sincronizar por si acaso. Una opción es aprovechar la semántica de java que sólo inicializa sus campos cuando la clase se llama por primera vez:

```

public class Singleton {
    private final static Singleton instancia = new Singleton();

    private Singleton() {
        // lo construye
    }

    public static Singleton getInstancia() {
        return instancia;
    }
}

```

Este código es correcto y eficaz pero tiene como pega que si hubiera más métodos en la clase, el campo se inicializaría con el primer método llamado, no necesariamente con `getInstancia()`. Si esto fuera un problema, podemos recurrir a un nivel más de encapsulamiento:

```

public class Singleton {
    private Singleton() {
        // lo construye
    }

    public static Singleton getInstancia() {
        return Aux.instancia;
    }

    private static class Aux {
        static final Singleton instancia = new Singleton();
    }
}

```

Diccionario

1. Acrónimos

API	Application Programming Interface
IDE	Integrated Development Environment
JDK	Java Development Kit
JRE	Java Runtime Environment
JVM	Java Virtual Machine
OS	Operating System
SDK	Software Development Kit
SO	Sistema Operativo (OS)
VM	Virtual Machine

2. Términos en inglés

<i>inglés</i>	<i>español</i>
atomic	<u>atómico, atómica</u>
concurrency	<u>conurrencia</u>
daemon	<u>demonio</u>
deadlock	<u>interbloqueo</u>
executor	<u>ejecutor</u>
exception	<u>excepción</u>
fairness	<u>equidad</u>
interrupt	<u>interrupción, interrumpir</u>
livelock	<u>interbloqueo activo</u>
liveness	<u>vivacidad</u>
lock	<u>cerrojo</u>
monitor	<u>monitor</u>
queue	cola
reentrant	<u>reentrante</u>
safety	<u>seguridad</u>
semaphore	<u>semáforo</u>

<i>inglés</i>	<i>español</i>
starvation	<u><i>inanición</i></u>
thread	<u><i>hebra, hilo</i></u>
thread-safe	<u><i>a prueba de concurrencia</i></u>
timer	<u><i>temporizador</i></u>
volatile	<u><i>volátil</i></u>