

Control de la Concurrencia en Java (API Estándar)

Tema 4 - Programación Concurrente y de Tiempo Real

Antonio J. Tomeu¹ Manuel Francisco²

¹Departamento de Ingeniería Informática
Universidad de Cádiz

²Departamento de CC. de la Computación e I.A.
Universidad de Granada

PCTR, 2019

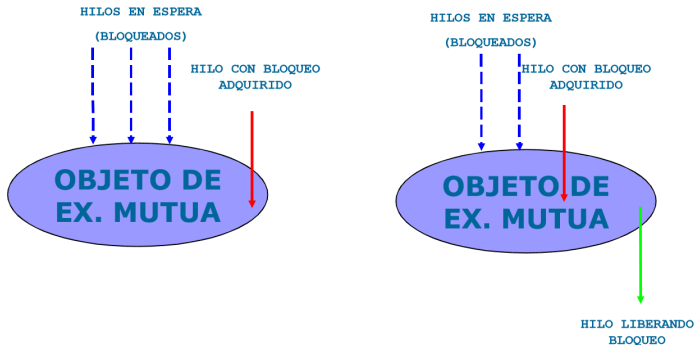
Contenido

1. Exclusión Mutua con código `synchronized`.
2. Exclusión Mutua con métodos `synchronized`.
3. Protocolos de Control de la Exclusión Mutua.
4. Interbloqueos.
5. Sincronización: `wait()`, `notify()` y `notifyAll()`.
6. Condiciones de Guarda.
7. Protocolos de Sincronización.
8. Diseño de monitores en Java.

Exclusión Mutua entre Hilos I

- ▶ En Java, la Exclusión Mutua se logra a **nivel de objetos**. Todo objeto tiene asociado un cerrojo (lock).
- ▶ Técnicas de Control de la Exclusión Mutua en Java:
 - ▶ **Bloques sincronizados**. El segmento de código que debe ejecutarse en exclusión mutua se etiqueta como **synchronized**. Es el equivalente a una región crítica en Java.
 - ▶ **Métodos sincronizados**. Aquellas instancias de recursos que deben manejarse bajo exclusión mutua se encapsulan en una clase y **todos** sus métodos modificadores son etiquetados con la palabra reservada **synchronized**.
- ▶ Por tanto, dos o más hilos están en exclusión mutua cuando llaman a métodos **synchronized** de un objeto (de exclusión mutua) o ejecutan métodos que tienen bloques de código sincronizados.

Exclusión Mutua entre Hilos II



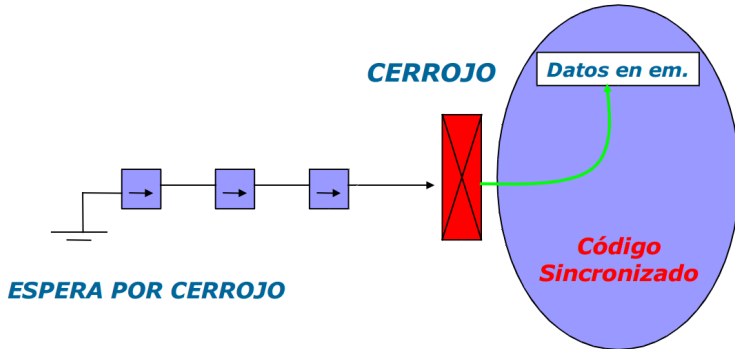
- ▶ Cuestiones a determinar:
 - ▶ ¿Cómo establecer el bloqueo a nivel sintáctico?
 - ▶ ¿Cuál es la semántica del bloqueo?
 - ▶ ¿Qué política de acceso se da a los hilos en espera?

Sintaxis para Bloques de Código Sincronizado

```
1 public metodo_x (parametros)
2 {
3     //Resto de codigo del metodo. No se ejecuta en EM.
4
5     /* comienzo de la seccion critica */
6     synchronized (object)
7     {
8         //Bloque de sentencias criticas
9         //Todo el codigo situado entre llaves se ejecuta bajo EM
10    }
11    /* fin de la seccion critica */
12 } //metodo_x
```

- ▶ Son una implementación del concepto de **región crítica**.
- ▶ Un bloque de código está sincronizado respecto de un objeto (puede ser el mismo objeto, `this`).
- ▶ El bloque sólo se ejecuta si se obtiene el cerrojo sobre el objeto.
- ▶ Útil para adecuar código secuencial a un entorno concurrente.

Política de Acceso de los Hilos en Espera



Acotando la Sección Crítica...

Código 1: codigos_t4/codBloqueo.java

```
1  public class codBloqueo //usa el cerrojo del propio objeto
2  {
3      private int numVueltas;
4
5      public codBloqueo(int vueltas)
6      {
7          numVueltas = vueltas;
8      }
9
10     public void metodo()
11     {
12         synchronized(this) //bloqueo para acceso a seccion critica
13         {
14             for(int i=1; i<=numVueltas; i++)
15                 System.out.print(i+" ");
16         }
17     }
18 }
```


... y Usándola desde Hilos en Modo Seguro I

Código 2: codigos_t4/UsacodBloqueo.java

```
1 public class UsacodBloqueo extends Thread {
2     codBloqueo cerrojo; //referencia a objeto compartido
3
4     public UsacodBloqueo(codBloqueo l) {
5         cerrojo = l;
6     }
7
8     public void run() {
9         cerrojo.metodo(); //metodo que tiene codigo sincronizado
10    }
11
12    public static void main(String[] args) {
13        codBloqueo aux = new codBloqueo(200);
14        UsacodBloqueo h1 = new UsacodBloqueo(aux);
15        UsacodBloqueo h2 = new UsacodBloqueo(aux);
16        h2.start();
17        h1.start();
18    }
19 }
```

Output con/sin synchronized

```
General Output
-----Configuration: <Default>-----
1 2 3 4 5 6 7 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48
49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100 101 102 103 104 105 106 107 108 109
110 111 112 113 114 115 116 117 118 119 120 121 122 123 124 125 126 127 128 129 130 131 132 133 134 135 136 137 138 139 140 141 142 143 144 145 146 147 148 149 150 151 152 153 154 155 156 157
158 159 160 161 162 163 164 165 166 167 168 169 170 171 172 173 174 175 176 177 178 179 180 181 182 183 184 185 186 187 188 189 190 191 192 193 194 195 196 197 198 199 200 24 25 2 27 28 29 30
31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94
95 96 97 98 99 100 101 102 103 104 105 106 107 108 109 110 111 112 113 114 115 116 117 118 119 120 121 122 123 124 125 126 127 128 129 130 131 132 133 134 135 136 137 138 139 140 141 142 143
144 145 146 147 148 149 150 151 152 153 154 155 156 157 158 159 160 161 162 163 164 165 166 167 168 169 170 171 172 173 174 175 176 177 178 179 180 181 182 183 184 185 186 187 188 189 190 191
... ..
General Output
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68
69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100 101 102 103 104 105 106 107 108 109 110 111 112 113 114 115 116 117 118 119 120 121 122 123 124
125 126 127 128 129 130 131 132 133 134 135 136 137 138 139 140 141 142 143 144 145 146 147 148 149 150 151 152 153 154 155 156 157 158 159 160 161 162 163 164 165 166 167 168 169 170 171 172
173 174 175 176 177 178 179 180 181 182 183 184 185 186 187 188 189 190 191 192 193 194 195 196 197 198 199 200 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30
31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94
95 96 97 98 99 100 101 102 103 104 105 106 107 108 109 110 111 112 113 114 115 116 117 118 119 120 121 122 123 124 125 126 127 128 129 130 131 132 133 134 135 136 137 138 139 140 141 142 143
144 145 146 147 148 149 150 151 152 153 154 155 156 157 158 159 160 161 162 163 164 165 166 167 168 169 170 171 172 173 174 175 176 177 178 179 180 181 182 183 184 185 186 187 188 189 190 191
192 193 194 195 196 197 198 199 200
Process completed
```

Pasando el Cerrojo como Parámetro I

Código 3: codigos_t4/MuestraBloqueo.java

```
1 public class MuestraBloqueo implements Runnable {
2     private Object o;
3
4     public MuestraBloqueo(Object p) {
5         o = p;
6     }
7
8     public void run() {
9         synchronized(o) {
10             for (int i = 1; i < 100; i++) {
11                 System.out.println("Iteracion " + i + " del hilo " +
12                     this.toString());
13                 for (int j = 1; j < 100; j++);
14             }
15         }
16     }
17 }
18
```

Pasando el Cerrojo como Parámetro II

```
19 public static void main(String[] args) {
20     Object lock = new Object();
21     Thread h1 = new Thread(new MuestraBloqueo(lock), "hilo 1");
22     Thread h2 = new Thread(new MuestraBloqueo(lock), "hilo 2");
23     h1.setPriority(Thread.MIN_PRIORITY);
24     h2.setPriority(Thread.MAX_PRIORITY);
25     h1.start();
26     h2.start();
27 }
28 }
```

- ▶ Descargue otra vez nuestra vieja clase `Cuenta_Banca.java`
- ▶ Decida que código debe ser sincronizado, y sincronícelo. La nueva clase será `Cuenta_Banca_Sync.java`
- ▶ Escriba un programa multihebrado que use objetos de la clase anterior. Llámelo `UsaCuenta_Banca_Sync.java`

Sintaxis para Métodos Sincronizados

```
1 public synchronized metodo_x (parametros)
2 {
3     //Bloque de sentencias del metodo
4     //Todo el codigo del metodo se ejecuta en em.
5 }
```

- ▶ Un método `synchronized` fuerza la exclusión mutua entre todos los hilos que lo invocan
- ▶ También fuerza la exclusión mutua con otros métodos `synchronized` del objeto
- ▶ Cualquier hilo que trate de ejecutar un método sincronizado deberá esperar si otro método sincronizado ya está en ejecución.
- ▶ Los métodos no sincronizados pueden seguir ejecutándose concurrentemente.

- ▶ El cerrojo está asociado a cada **instancia** de una clase (objeto)
- ▶ Los métodos de clase (`static`) también pueden ser `synchronized`
- ▶ En clases heredadas, los métodos sobrescritos pueden ser sincronizados o no, sin afectar a cómo era (y es) el método de la superclase

Control con Métodos Sincronizados I

Código 4: codigos_t4/MuestraBloqueoObjeto.java

```
1 public class MuestraBloqueoObjeto {
2
3     public synchronized void metodoA() {
4         for (int i = 1; i < 100; i++) {
5             System.out.println("Iteracion " + i + " del metodo A ");
6             for (int j = 1; j < 100; j++);
7         }
8         System.out.println("metodo A liberando cerrojo...");
9     }
10
11     public synchronized void metodoB() {
12         for (int i = 1; i < 100; i++) {
13             System.out.println("Iteracion " + i + " del metodo B ");
14             for (int j = 1; j < 100; j++);
15         }
16         System.out.println("metodo B liberando cerrojo...");
17     }
18
19 }
```

Control con Métodos Sincronizados II

Código 5: codigos_t4/UsaMuestraBloqueoObjeto.java

```
1  public class UsaMuestraBloqueoObjeto implements Runnable {
2      private MuestraBloqueoObjeto p;
3      private int caso;
4      public UsaMuestraBloqueoObjeto(MuestraBloqueoObjeto o, int
        val) {
5          p = o;
6          caso = val;
7      }
8
9      public void run() {
10         switch (caso) {
11             case 0:
12                 p.metodoA();
13                 break;
14             case 1:
15                 p.metodoB();
16                 break;
17         }
18     }
19
20     public static void main(String[] args) {
```

Control con Métodos Sincronizados III

```
21     MuestraBloqueoObjeto monitor = new MuestraBloqueoObjeto();
22     Thread h1 = new Thread(new UsaMuestraBloqueoObjeto(monitor,
23         0));
24     Thread h2 = new Thread(new UsaMuestraBloqueoObjeto(monitor,
25         1));
26     h1.start();
27     h2.start();
28 }
```

Código 6: codigos_t4/ExMutua.java

```
1  /*Ejemplo de Exclusion Mutua entre Hilos
2   *@author Antonio J. Tomeu
3   *Varios hilos concurrentes modifican un contador protegido
4   *    bajo e.m.
5   *El metodo que incrementa al contador es synchronized
6   *Crea un array de hilos que incrementan bajo e.m. el contador
7   *Sintaxis de uso: java ExMutua n m donde:
8   *n es el numero de hilos concurrentes
9   *m es el valor inicial del contador
10  */
11  class ObCritico {
12      private int Dato; //Contiene el objeto critico
13
14      public ObCritico(int VInicial) //el constructor
15      {
16          Dato = VInicial;
17      }
18
19      public synchronized void Incremento() //e ejecutar bajo e.m.
20      {
21          Dato++;
22      }
```

```
23
24     public int Valor() //hace una lectura. No necesita e.m.
25     {
26         return (Dato);
27     }
28 }
29
30 public class ExMutua extends Thread {
31     private ObCritico SC;
32
33     public ExMutua(ObCritico SecCritica) {
34         SC = SecCritica;
35     }
36
37     public void run() {
38         for (;;) {
39             SC.Incremento();
40             System.out.println("El hilo con id. " + getName() + " ha
                ajustado el objeto a valor " + SC.Valor());
41         }
42     }
43
44     public static void main(String[] args) {
45         if (args.length != 2) {
46             System.err.println("Sintaxis: java ExMutua n m");
```

```
47     System.exit(1);
48 }
49
50 int NumHilos = Integer.valueOf(args[0]).intValue(); //fija
    numero de hilos
51 ObCritico ContadorCritico = new
    ObCritico(Integer.valueOf(args[1]).intValue());
52 ExMutua[] Hilos = new ExMutua[NumHilos];
53 for (int i = 0; i <= NumHilos - 1; i++) {
54     Hilos[i] = new ExMutua(ContadorCritico);
55 }
56 for (int i = 0; i <= NumHilos - 1; i++) {
57     Hilos[i].start();
58 }
59 }
60 }
```

- ▶ Descargue otra vez nuestra vieja clase `Cuenta_Banca.java`
- ▶ Decida qué métodos deben ser sincronizados, y sincronícelos. La nueva clase será `Cuenta_Banca_Sync_Met.java`
- ▶ Escriba un programa multihebrado que use objetos de la clase anterior. Llámelo `Usa_Cuenta_Banca_Sync_Met.java`

- ▶ Acceder al recurso compartido donde sea necesario
- ▶ Definir un objeto para control de la exclusión mutua (o usar el propio objeto, `this`).
- ▶ Sincronizar el código crítico utilizando el cerrojo del objeto de control dentro de un bloque `synchronized` de instrucciones.

- ▶ Encapsular el recurso crítico en una clase
- ▶ Definir todos los métodos de la clase como no estáticos y `synchronized`
- ▶ O, al menos, todos los modificadores
- ▶ Crear hilos que compartan una instancia de la clase creada
- ▶ ¡OJO! Esto no provee sincronización

- ▶ Modele con una clase alguna situación donde haya presencia de concurrencia y condiciones de concurso. Guárdela en `Recurso.java`
- ▶ Modifique la clase anterior protegiendo al recurso mediante el Segundo Protocolo de Control de la E.M. Llame a la clase `Recurso_em.java`
- ▶ Escriba ahora un programa llamado `Prueba_Recurso_em.java` que haga uso de la clase anterior. Debe crear varios hilos concurrentes mediante implementación de la interfaz `Runnable`.

La Posesión del Bloqueo es por Hilo...

- ▶ Un método sincronizado puede invocar a otro método sincronizado sobre el mismo objeto (REENTRANCIA)
- ▶ Permite llamadas recursivas a métodos sincronizados
- ▶ Permite invocar métodos heredados sincronizados
- ▶ Descargue `obj_protected.java` y `usa_obj_protected.java`. Pruébelos. ¿A qué conclusión llega?
- ▶ Incorpore ahora un método `m3()` recursivo dentro de `obj_protected.java`. ¿Se mantiene su conclusión?

Interbloqueo (Deadlock) Entre Hilos I

- ▶ Se producen cuando hay condiciones de espera de liberación de bloqueos cruzados entre dos hilos
- ▶ Sólo pueden evitarse mediante un análisis cuidadoso y riguroso del uso de código sincronizado

Interbloqueo (Deadlock) Entre Hilos II

Código 7: codigos_t4/Deadlock.java

```
1 public class Deadlock {
2     public static void main(String[] args) {
3
4         final Object region_A = new Object();
5         final Object region_B = new Object();
6
7         Thread Hilo_A = new Thread(new Runnable() {
8             public void run() {
9                 synchronized(region_A) {
10                     synchronized(region_B) {
11                         System.out.println("hilo A");
12                     }
13                 }
14             }
15        });
16
17        Thread Hilo_B = new Thread(new Runnable() {
18            public void run() {
19                synchronized(region_B) {
20                    synchronized(region_A) {
21                        System.out.println("hilo B");
```

Interbloqueo (Deadlock) Entre Hilos III

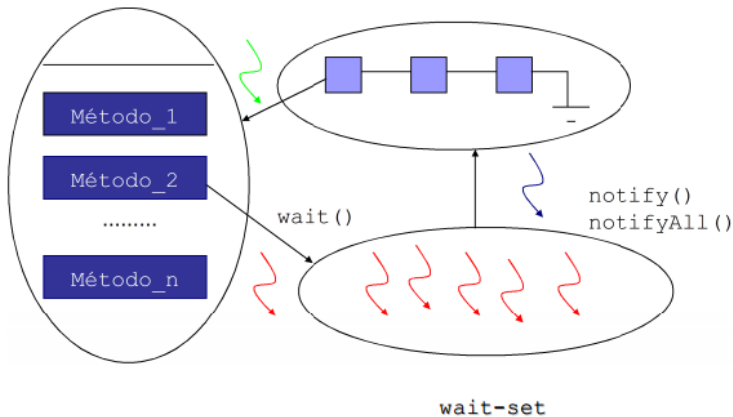
```
22     }
23     }
24     }
25     });
26
27
28     Hilo_B.start();
29     Hilo_A.start();
30 }
31 }
```

Sincronización Entre Hilos

- ▶ En Java la sincronización entre hilos se logra con los métodos `wait`, `notify` y `notifyAll` (clase `Object`).
- ▶ Deben ser utilizados únicamente **dentro** de métodos o código de tipo `synchronized`
- ▶ Cuando un hilo llama a un método que hace `wait` las siguientes acciones son ejecutadas **atómicamente**:
 1. Hilo llamante **suspendido y bloqueado**
 2. Exclusión mutua sobre el objeto **liberada**
 3. Hilo colocado en una **cola única (el wait-set)** de espera asociada al objeto
- ▶ Cuando un hilo llama a un método que hace `notify`, uno de los hilos bloqueados en la cola pasa a listo. **Java no especifica cuál**. Depende de la implementación (JVM). Si se llama al método `notifyAll`, **todos** los hilos de dicha cola son desbloqueados y pasan a listo. Accederá al objeto aquél que sea planificado.
- ▶ Si el wait-set está vacío, `notify` y `notifyAll` no tienen efecto.

El Conjunto de Espera (wait-set)

Conjunto de espera asociado al cerrojo e.m del objeto



Código 8: codigos_t4/aDormir.java

```
1 public class aDormir extends Thread {
2     public aDormir() {}
3     public void run() {
4         System.out.println("El hilo " + this.getName() + " dijo: mi
5             vida activa fue breve...");
6         synchronized(this) {
7             try {
8                 wait();
9             } catch (InterruptedException e) {} //cada hilo dormido
10                sobre su propio cerrojo
11            System.out.println(this.getName() + " dijo: pero he
12                revivido...");
13        }
14    }
15
16    public void despertar() {
17        synchronized(this) {
18            notify();
19        }
20    }
21
22    public static void main(String[] args)
23        throws Exception {
```

```

21     aDormir[] h = new aDormir[10];
22     for (int i = 0; i < 10; i++) {
23         h[i] = new aDormir();
24         h[i].start();
25     }
26     try {
27         Thread.currentThread().sleep(1000);
28     } catch (InterruptedException e) {}
29     Thread.currentThread().sleep(2000); //retardo evita perdida
        de senial...
30     h[5].despertar();
31 }
32 }

```

Código 9: codigos_t4/aDormir2.java

```

1 public class aDormir2 extends Thread {
2     Object lock;
3     public aDormir2(Object l) {
4         lock = l;
5     }
6     public void run() {
7         System.out.println("El hilo " + this.getName() + " dijo: mi
            vida activa fue breve...");
8         synchronized(lock) {

```

```
9      try {
10         lock.wait();
11     } catch (InterruptedException e) {} //cada hilo dormido
        sobre su propio cerrojo
12     System.out.println(this.getName() + " dijo: pero he
        revivido...");
13 }
14 }
15
16 public void despertar() {
17     synchronized(lock) {
18         lock.notify();
19     }
20 }
21 public void despertarTodos() {
22     synchronized(lock) {
23         lock.notifyAll();
24     }
25 }
26
27 public static void main(String[] args) {
28     Object cerrojo = new Object();
29     aDormir2[] h = new aDormir2[10];
30     for (int i = 0; i < 10; i++) {
31         h[i] = new aDormir2(cerrojo);
```

```
32     h[i].start();
33 }
34 h[5].despertar();
35 h[5].despertarTodos();
36 System.out.print("Todos terminaron...");
37
38 }
39 }
```

- ▶ No se puede escribir código de hilos asumiendo que un hilo concreto recibirá la notificación
- ▶ Diferentes hilos pueden estar bloqueados sobre un mismo objeto a la espera de diferentes condiciones, pero el wait-set es único
- ▶ Dado que `notifyAll()` los despierta a todos de forma incondicional, es posible que reactive hilos para los cuales no se cumple aún la condición de espera
- ▶ Solución: siempre que el hilo usuario invoca a `wait()`, lo primero al despertar es volver a comprobar su condición particular, volviendo al bloqueo si ésta aún no se cumple

Patrón de Código para Condiciones de Guarda

```
1 public synchronized void m_receptor_senal() {
2     //hilo usuario se bloqueara en el metodo a espera de
3     condicion
4     //el hilo usuario pasa al wait-set
5     while (!condicion) try {
6         wait();
7     } //condicion de guarda
8     catch ()(Exception e) {}
9     //codigo accedido en e.m.
10 }
11
12 public synchronized void m_emisor_senal() {
13     //hilo usuario enviara una senal a todos los hilos del waitset
14     //la guarda garantiza que se despiertan los adecuados
15     //codigo en e.m. que cambia las condiciones de estado
16     notifyAll();
17 }
```

Protocolo de Sincronización Inter-Hilos en Java con Espera Ocupada

```
1  // Thread A
2  public void waitForMessage() {
3      while (hasMessage == false) {
4          Thread.sleep(100);
5      }
6  }
7  // Thread B
8  public void setMessage(String message) {
9      ...
10     hasMessage = true;
```

Protocolo de Sincronización Inter-Hilos en Java con Sincronización wait-notify

```
1 // Thread A
2 public synchronized void waitForMessage() {
3     try {
4         wait();
5     } catch (InterruptedException ex) {}
6 }
7 // Thread B
8 public synchronized void setMessage(String message) {
9     ...
10    notify();
```

- ▶ OJO: No es perfecto. La señal puede perderse (y se pierde) si nadie está esperándola.
- ▶ SOLUCIÓN: Utilizar condiciones. A sólo se bloqueará si al comprobar una condición es falsa, y B se asegurará de hacer que la condición sea verdadera antes de notificar.

Código 10: codigos_t4/Sincronizacion.java

```
1  class Sincro
2
3  {
4      int Turno;
5
6      public Sincro(int t) {
7          Turno = t;
8      }
9
10     public synchronized void metodo1() {
11         while (Turno != 1)
12             try {
13                 wait();
14             } catch (Exception e) {}
15         System.out.println(" Turno al hilo1...");
16         Turno = 2;
17         notifyAll();
18     }
19 }
```

Patrón "Sincronizador" II

```
20 public synchronized void metodo2() {
21     while (Turno != 2)
22         try {
23             wait();
24         } catch (Exception e) {}
25     System.out.println("Turno al hilo 2...");
26     Turno = 3;
27     notifyAll();
28 }
29
30
31 public synchronized void metodo3() {
32     while (Turno != 3)
33         try {
34             wait();
35         } catch (Exception e) {}
36     System.out.println("Turno al hilo 3...");
37     Turno = 1;
38     notifyAll();
39 }
40
41 } //Sincro
```

Patrón "Sincronizador" III

```
42
43 class Hilo1 extends Thread {
44     Sincro ref;
45     public Hilo1(Sincro obj) {
46         ref = obj;
47     }
48     public void run() {
49         for (;;) {
50             ref.metodo1();
51         }
52     }
53 } //Hilo1
54
55
56 class Hilo2 extends Thread {
57     Sincro ref;
58     public Hilo2(Sincro obj) {
59         ref = obj;
60     }
61     public void run() {
62         for (;;) {
63             ref.metodo2();
```

Patrón "Sincronizador" IV

```
64     }
65     }
66 } //Hilo2
67
68 class Hilo3 extends Thread {
69     Sincro ref;
70     public Hilo3(Sincro obj) {
71         ref = obj;
72     }
73     public void run() {
74         for (;;) {
75             ref.metodo3();
76         }
77     }
78 } //Hilo3
79
80
81
82
83 public class Sincronizacion {
84     public static void main(String[] args) {
85         Sincro m = new Sincro(2);
```

Patrón "Sincronizador" V

```
86     new Hilo1(m).start();
87     new Hilo2(m).start();
88     new Hilo3(m).start();
89     System.out.println("hilos lanzados...");
90
91 } //main
92 } //Sincronizacion
```

- ▶ Utilizando condiciones de guarda, modele un protocolo de sincronización simple Hilo B \rightarrow Hilo A que sea completamente correcto.
- ▶ Guarde su código en ficheros S1.java, S2.java...

- ▶ Un monitor es un objeto que implementa acceso bajo exclusión mutua a **todos** sus métodos, y provee sincronización
- ▶ En Java, son objetos **de una clase cuyos métodos públicos son todos synchronized**
- ▶ Un objeto con métodos `synchronized` proporciona un cerrojo único que permite implantar monitores con comodidad y exclusión mutua bajo control
- ▶ Los métodos `wait()`, `notify()` y `notifyAll()` permiten sincronizar los accesos al monitor, de acuerdo a la semántica de los mismos ya conocida.

Estructura Sintáctica de un Monitor en Java

```
1  class Monitor {
2      //definir aquí datos protegidos por el monitor
3      public Monitor() {...} //constructor
4      public synchronized tipo1 metodo1()
5      throws InterruptedException {
6          ...
7          notifyAll();
8          ...
9          while (!condicion1) wait();
10     }
11     public synchronized tipo2 metodo2()
12     throws InterruptedException {
13         ...
14         notifyAll();
15         ...
16         while (!condicion1) wait();
17     }
18 }
```


Semántica de un Monitor en Java I

- ▶ Cuando un método `|synchronized|` del monitor llama a `wait()`, libera la exclusión mutua sobre el monitor y encola al hilo que llamó al método en el wait-set.
- ▶ Cuando otro método del monitor hace `notify()`, un hilo del wait-set (Java no especifica cuál) pasará a la cola de hilos que esperan el cerrojo y se reanudará cuando sea planificado.
- ▶ Cuando otro método del monitor hace `notifyAll()`, todos los hilos del wait-set pasarán a la cola de hilos que esperan el cerrojo y se reanudarán cuando sean planificados.
- ▶ El monitor Java **no tiene variables de condición, sólo una cola de bloqueo de espera implícita**
- ▶ La política de señalización es señalar y seguir (SC)
 - ▶ El método (hilo) señalador sigue su ejecución
 - ▶ El hilo(s) señalado(s) pasan del wait-set a la cola de procesos que esperan el cerrojo

- ▶ Para dormir a un hilo a la espera de una condición usamos el método `wait()` (dentro de un método `synchronized`)
- ▶ Es menos fino que una variable de condición
- ▶ El conjunto de espera para `wait` es único (wait-set)

Monitores en Java: Peculiaridades

- ▶ No es posible programar a los hilos suponiendo que recibirán la señalización cuándo la necesiten.
- ▶ Al no existir variables de condición, sino una única variable implícita, es conveniente usar `notifyAll()` para que todos los procesos comprueben la condición que los bloqueó.
- ▶ No es posible señalar un hilo en especial, por tanto:
 - ▶ Es aconsejable bloquear a los hilos en el wait-set con una condición de guarda en conjunción con `notifyAll()`.
 - ▶

```
while (!condicion) try {wait();}  
    catch (InterruptedException e) {return;}
```
 - ▶ Todos serán despertados, comprobarán la condición y volverán a bloquearse, excepto los que la encuentren verdadera (que pasan a espera del cerrojo sobre el monitor).
- ▶ No es aconsejable comprobar la condición de guarda con `if`
- ▶ Los campos protegidos por el monitor suelen declararse `private`

1. Decidir qué datos encapsular en el monitor
2. Construir un monitor teórico, utilizando **tantas variables de condición como sean necesarias**
3. Usar la señalización SC en el monitor teórico
4. Implementar en Java
 - 4.1 Escribir un método synchronized por cada procedimiento
 - 4.2 Hacer los datos encapsulados private
 - 4.3 Sustituir cada wait(variableCondicion) por una condición de guarda.
 - 4.4 Sustituir cada send(variableCondicion) por una llamada a notifyAll()
 - 4.5 Escribir el código de inicialización del monitor en el constructor del mismo

Código 11: codigos_t4/MonitorSimple.java

```
1  /**Ejemplo de Monitor sencillo. Encapsula una variable
    protegida por
2   *la abstraccion y posee una interfaz de dos metodos para
3   *incrementarla y decrementarla y un tercero para conocer el
    valor
4   *del recurso protegido.
5   *@author Antonio Tomeu
6   */
7  class Monitor {
8      private static int Dato; //recurso protegido
9      public Monitor(int VInic) {
10         Dato = VInic;
11     }
12     public synchronized void INC() {
13         while (!(Dato <= 0))
14             try {
15                 System.out.println("Hilo Sumador bloqueado");
16                 wait();
17             } catch (InterruptedException e) {}
18         Dato++;
19         notifyAll();
20     }
21 }
```

```
22 public synchronized void DEC() {
23     while (!(Dato > 0))
24         try {
25             System.out.println("Hilo Restador bloqueado");
26             wait();
27         } catch (InterruptedException e) {}
28     Dato--;
29     notifyAll();
30 }
31 public synchronized String toString() {
32     return (new Integer(Dato).toString());
33 }
34 }
35 class HiloSumador
36 extends Thread {
37     private Monitor Data;
38     public HiloSumador(Monitor Ref) {
39         Data = Ref;
40     }
41     public void run() {
42         for (;;) Data.INC();
43     }
44 }
45
46 class HiloRestador
```

```
47 extends Thread {
48     private Monitor Data;
49     public HiloRestador(Monitor Ref) {
50         Data = Ref;
51     }
52     public void run() {
53         for (;;) Data.DEC();
54     }
55 }
56 public class MonitorSimple {
57     public static void main(String[] args) {
58         Monitor O = new Monitor(1000);
59         new HiloSumador(O).start();
60         new HiloRestador(O).start();
61         new HiloRestador(O).start();
62         new HiloRestador(O).start();
63         for (;;) System.out.println(O.toString());
64     }
65 }
```

Código 12: codigos_t4/Buffer.java

```
1  /**Ejemplo de Monitor sencillo para productor-consumidor.  
2   *Encapsula un buffer protegida por la abstraccion  
3   *y posee una interfaz de dos metodos para insertar y extraer,  
4   *y se provee  
5   *la sincronizacion necesaria. Observe que la condicion de  
6   *guarda es de la  
7   *forma if(...)try{wait()}... en lugar de  
8   *while(!condicion)try{wait()}... De  
9   *igual forma se senializa con notify(), en lugar de con  
10  *notifyAll().  
11  *Funcionaria esto con varios productores y consumidores?  
12  *@author Antonio Tomeu  
13  */  
14  
15  public class Buffer {  
16      private int numSlots = 0;  
17      private double[] buffer = null;  
18      private int putIn = 0, takeOut = 0;  
19      private int cont = 0;
```


Monitor Productor-Consumidor II

```
16
17     public Buffer(int numSlots) {
18         this.numSlots = numSlots;
19         buffer = new double[numSlots];
20     }
21
22     public synchronized void insertar(double valor) {
23         while (cont == numSlots)
24             try {
25                 wait();
26             } catch (InterruptedException e) {
27                 System.err.println("wait interrumpido");
28             }
29         buffer[putIn] = valor;
30         putIn = (putIn + 1) % numSlots;
31         cont++;
32         notifyAll();
33     }
34
35     public synchronized double extraer() {
36         double valor;
37         while (cont == 0)
```

Monitor Productor-Consumidor III

```
38     try {
39         wait();
40     } catch (InterruptedException e) {
41         System.err.println("wait interrumpido");
42     }
43     valor = buffer[takeOut];
44     takeOut = (takeOut + 1) % numSlots;
45     cont--;
46     notifyAll();
47     return valor;
48 }
49 } //Buffer
```

Código 13: codigos_t4/Prueba_Prod_Con.java

```
1 class Productor implements Runnable {
2
3     private Buffer bb = null;
4
5     public Productor(Buffer bb) {
6         this.bb = bb;
7     }
}
```

Monitor Productor-Consumidor IV

```
8
9     public void run() {
10         double item = 0.0;
11         while (true) {
12             bb.insertar(++item);
13             System.out.println("Produciendo " + item);
14         }
15     }
16 } //Productor
17
18 class Consumidor implements Runnable {
19
20     private Buffer bb = null;
21     public Consumidor(Buffer bb) {
22         this.bb = bb;
23     }
24
25     public void run() {
26         double item;
27         while (true) {
28             item = bb.extraer();
29             System.out.println("Consumiendo " + item);
```

Monitor Productor-Consumidor V

```
30     }
31 }
32
33 } //Consumidor
34
35 public class Prueba_Prod_Con {
36     public static void main(String[] args) {
37         int ranuras = 10;
38         Buffer monitor = new Buffer(ranuras);
39
40         new Thread(new Productor(monitor)).start();
41         new Thread(new Consumidor(monitor)).start();
42
43     } //main
44 } //Prueba_Prod_Con
```

- ▶ Aumente el número de lectores y escritores, y verifique la sincronización
- ▶ Diseñe un monitor en Java que modele la abstracción semáforo, dótelo de métodos wait y signal y guárdelo en `semaforo.java`
- ▶ Escriba ahora un protocolo de exclusion mutua con una instancia de la clase anterior, y guárdelo en `Usa_semaf.java`, que utilice la cuenta encapsulada del monitor.
- ▶ Construya un monitor que dé solución al problema de los filósofos



Göetz et al.

Java Concurrency in Practice

2006



Oaks & Wong.

Java Threads

O'Reilly, 2004



Wellings, A.

Concurrent and Real Time Programming in Java.

John Wiley & Sons, 2004