ACTIVIDADES DE AUTOCORRECCIÓN - UNIDADES 1, 2 Y 3

ACTIVIDAD 1 – Programación concurrente

1.1 Describa las ventajas y los inconvenientes de la programación concurrente frente a la programación secuencial.

1.2 Describa las propiedades que garantizan la corrección de un programa concurrente.

ACTIVIDAD 2 – Cooperación entre hilos

2.1 Indique cuáles son los dos mecanismos de comunicación que se pueden utilizar para permitir la comunicación entre hilos o procesos. Explique cuáles son las características más relevantes de dichos mecanismos.

2.2 Justifique si la comunicación entre actividades debería estar complementada por un mecanismo de sincronización para facilitar la colaboración entre actividades.

2.3 Describa qué se entiende por **acción atómica** y **estado consistente** de un objeto.

2.4 Describa qué es una **sección crítica** y qué propiedades deben garantizar los protocolos que regulan el acceso a las secciones críticas.

ACTIVIDAD 3 – Hilos en Java

3.1 Dibuje el ciclo de vida de los hilos en Java, indicando: (i) en qué estado puede estar un hilo en cada momento; (ii) qué evento o eventos permiten pasar de un estado a otro.

3.2 Explique brevemente qué realiza cada uno de los siguientes métodos de la clase Thread de Java:

run()	start()
join()	yield()
isAlive()	sleep()
interrupt()	

3.3 Existen diferentes formas de definir hilos en Java. Podemos clasificarlas según: (i) se cree una clase con nombre, que permita declarar diferentes instancias, o bien se cree una clase anónima; (ii) se extienda de la clase Thread o se implemente la interfaz Runnable.

A continuación se muestra un ejemplo de creación de un hilo:

```
public class NewThread{
    public static void main(){
     new Thread("Hilo") {
       public void run() {
        System.out.println("Esto es un hilo."); }
       }.start();
    }
}
```

a) Indique a qué clasificación corresponde el ejemplo anterior.

b) Modifique el ejemplo de forma oportuna para dar nuevos ejemplos del resto de clasificaciones.

ACTIVIDAD 4 - Determinismo

Dado el siguiente código:

```
public class Counter
{  protected long count=0;

   public void add(long x){
       Thread.currentThread().yield();
       long value=count;
       value+=x;
       Thread.currentThread().yield();
       count=value;

   }
  public long getCount() {
      return count; }
}


public class RaceCondition
{
   public static void main(String[] args)
  {
     Counter c = new Counter();
     MyThread tA = new MyThread(c, 'A', 3);
     //A adds 3
     MyThread tB= new MyThread(c, 'B', 2);
     //B adds 2

    System.out.println("----------------");

     tA.start();
     tB.start();

     }
}
```

```
public class MyThread extends Thread
{
    private Counter c;
    private char threadname;
    private long i;
    public MyThread(Counter counter, char name,
long adding)
    {
         c=counter;
         threadname = name;
         i=adding;
    }

  public void run()
   {   c.add(i);
       Thread.yield();
       System.out.println("Thread #" +
threadname + " adds: " + i);
       try {
            sleep((int)(Math.random() * 100));
           } catch (InterruptedException e) { }

       long value=c.getCount();
       System.out.println("Thread #" +
threadname + " reads: " + value);
    }
}
```

4.1 Describa brevemente qué realiza este código. Detalle para qué sirve cada una de las clases que contiene.

4.2 ¿Este código es determinista? Muestre varias trazas de ejecución del código para justificar su respuesta.

4.3 ¿Se pueden producir condiciones de carrera? Justifique su respuesta (por ejemplo, indicando cuál o cuáles son los objetos compartidos que provocan problemas, o bien con trazas de ejecución).

4.4 Indique dónde están las secciones críticas en el código anterior y cómo se deben proteger en Java.

ACTIVIDAD 5 – Sincronización

Se ha definido una clase *Counter* que permite realizar incrementos y decrementos sobre un contador. Además, en la clase *RaceCondition* se dispone del método principal *main*() en el que se lanzan cuatro hilos (2 incrementadores y 2 decrementadores), que actúan sobre la misma variable contador. Se requiere que el hilo principal imprima el valor final del contador tras la ejecución de los otros cuatro hilos.

A continuación se muestra el código de las clases *Counter*, *Incrementer*, *Decrementer* y *RaceCondition*.

```
class Counter {
    private int c = 0;

    public void increment() {
        c++;
    }

    public void decrement() {
        c--;
    }

    public int value() {
        return c;
    }
}
```

```
public class RaceCondition
{ public static void main(String[] args)
     { Counter c = new Counter();
       int loops=1000;
       System.out.println("Loops "+ loops );

       Incrementer inc1 = new Incrementer(c, 1, loops);
       Incrementer inc2 = new Incrementer(c, 2, loops);

       Decrementer dec1 = new Decrementer(c, 1, loops);
       Decrementer dec2 = new Decrementer(c, 2, loops);

       inc1.start(); inc2.start();
       dec1.start(); dec2.start();

       System.out.println("Final result: "+c.value() );
     }
}
```

```
public class Incrementer extends Thread
{  private Counter c;
   private int myname;
   private int cycles;
   public Incrementer(Counter count,
         int name, int quantity)
   {   c = count;
       myname = name;
       cycles=quantity;
   }

   public void run()
   { for (int i = 0; i < cycles; i++)
     { c.increment();
       try
       { sleep((int)(Math.random() * 10));
       } catch (InterruptedException e){ }
      }
    System.out.println("Incrementer #" + myname+"
has done "+cycles+" increments.");
   }
}
```

```
public class Decrementer extends Thread
{  private Counter c;
   private int myname;
   private int cycles;
   public Decrementer(Counter count,
        int name, int quantity)
   {   c = count;
       myname = name;
       cycles=quantity;
   }

   public void run()
   { for (int i = 0; i < cycles; i++)
     {  c.decrement();
        try
        { sleep((int)(Math.random() * 20));
        } catch (InterruptedException e) { }
     }
     System.out.println("Decrementer #" + myname + "
has done "+cycles+" decrements.");
   }
}
```

5.1 Explique qué tipos de sincronización son requeridos y por qué. Es decir, indique si:

a) ¿Se requiere exclusión mútua? ¿Por qué? ¿Dónde?

b) ¿Se requiere sincronización condicional? ¿Por qué?

c) ¿Se requiere algún otro tipo de sincronización entre los hilos? Si es así, ¿entre qué hilos?

5.2 Modifique a continuación el código inicial proporcionado para que se ofrezca una solución que garantice la **exclusión mutua** y la **sincronización entre los hilos.**

ACTIVIDAD 6 – Programación concurrente

Dado este programa en Java, en el que la columna que queda a la izquierda de cada línea de código proporciona su número para poder referenciar tales líneas en los ejercicios que siguen:

```
1.1   public class Buffer {
1.2     private int store = 0;
1.3     private boolean full = false;
1.4
1.5     public int get()   {
1.6         int value = store;
1.7         store = 0;
1.8         full = false;
1.9         return value;
1.10    }
1.11
1.12    public void put(int value)    {
1.13        full = true;
1.14        store = value;
1.15    }
1.16  }

2.1   public class Main {
2.2     public static void main(String[] args)  {
2.3         Buffer c = new Buffer();
2.4         Consumer c1 = new Consumer(c, 1);
2.5         Producer p1 = new Producer(c, 2);
2.6         c1.start();
2.7         p1.start();
2.8         System.out.println("Producer and " +
2.9           "Consumer have terminated.");
2.10    }
2.11  }
```

```
3.1   public class Consumer extends Thread {
3.2     private Buffer b;
3.3     private int number;
3.4     public Consumer(Buffer ca, int id)  {
3.5       b = ca;
3.6       number = id;
3.7     }
3.8     public void run()  {
3.9       int value = 0;
3.10      for (int i = 1; i < 101; i++)
3.11      {
3.12        value = b.get();
3.13        System.out.println("Consumer #" + number +
3.14        " gets: " + value);
3.15      }
3.16    }
3.17  }

4.1   public class Producer extends Thread {
4.2     private Buffer b;
4.3     private int number;
4.4     public Producer(Buffer ca, int id) {
4.5       b = ca;
4.6       number = id;
4.7     }
4.8     public void run()  {
4.9       for (int i = 1; i < 101; i++)
4.10      {
4.11        b.put(i);
4.12        System.out.println("Producer #" + number +
4.13        " puts: " + i);
4.14      }
4.15    }
4.16  }
```

6.1 Indique si este programa es ***determinista*** o no. Explique por qué. Para apoyar su explicación, puede dar ejemplos de traza y/o indicar las líneas de código que validen la respuesta proporcionada.

6.2 Indique si en este programa puede haber ***condiciones de carrera***. Para apoyar su explicación, puede dar ejemplos de traza y/o indicar las líneas de código que validen la respuesta proporcionada.

6.3 Indique si en este programa se requiere implementar alguno de estos tipos de sincronización: ***exclusión mutua*** y ***sincronización condicional***. Explique por qué, indicando:

a) El objeto u objetos compartidos que deberían ser accedidos en exclusión mutua, en su caso.

b) El método u métodos donde se debería tener en cuenta la sincronización condicional. Para dichos métodos, rellene también la siguiente tabla, que indica:

b1) en qué casos no se puede aplicar directamente el método, pues la condición que se requiere de sincronización es falsa, de modo que el hilo que lo ejecuta debe quedar suspendido, en espera.

b2) en qué casos, al modificar el método el estado de un determinado objeto, debe avisar a un hilo en espera de dicha modificación.

Método	Espera cuando	Avisa a

6.4. Indique qué cambios deberían realizarse sobre el programa presentado en el enunciado para implantar tanto la *exclusión mutua* como la *sincronización condicional* necesaria para que la clase Buffer se comporte como un buffer con capacidad para un solo elemento. Con ello, si un productor intenta insertar un elemento pero ya hubiese alguno en el buffer, el productor permanecería suspendido hasta que hubiere espacio. Por su parte, el consumidor debería suspenderse si al intentar extraer un elemento no hubiese ninguno en el buffer.

6.5 Indique qué código se debería añadir y dónde al programa presentado para que el hilo inicial escriba su último mensaje una vez hayan terminado los demás hilos.

ACTIVIDAD 7 – Variantes de monitor

7.1 Explique por qué se han definido diferentes variantes de monitor.

7.2 Marque qué características cumplen cada una de las variantes de monitor que se muestran en la siguiente tabla:

	Modelo o Variante de Monitor		
Característica	**Brinch Hansen**	**Hoare**	**Lampson & Redell**
El modelo necesita una cola especial de entrada			
El modelo obliga a que toda invocación a *notify()* sea la última sentencia en los métodos del monitor en los que aparezca tal invocación.			
En caso de que haya hilos suspendidos en alguna condición, suspende al hilo que ha invocado a *notify*(), activando a uno de los hilos que llamó antes a *wait*().			
No reactiva a ningún hilo suspendido por haber llamado a *wait*() mientras el hilo notificador no abandone voluntariamente el monitor.			
El modelo jamás suspende al hilo que invoque a *notify*().			
El modelo suspende (y deja en una cola especial de entrada) al hilo que ha invocado a *notify*(), activando a uno de los hilos que llamó antes a *wait*().			
El modelo exige que todo método en el que se use *notify*() también tenga alguna llamada a *wait*().			

ACTIVIDAD 8 – Monitores en Java

El siguiente código Java pretende implantar un monitor para controlar el acceso de hilos lectores (concurrentes) y escritores (de manera exclusiva) sobre un recurso compartido. Los escritores utilizarán *writeStart*() antes de acceder y *writeEnd*() tras haber accedido. Los lectores utilizarán *readStart*() antes de acceder y *readEnd*() tras haber accedido:

```
public class ReadersWriters {
   private int writersWaiting;
   private boolean writing;
   private int readers;

   public ReadersWriters() {
       writersWaiting=readers=0;
       writing=false;
   }

   public synchronized void readStart() {
       while (writing || writersWaiting>0)
          try { wait(); }
            catch(Exception e) { };
       readers++;
   }
```

```
    public synchronized void readEnd() {
        readers--;
        notifyAll();
    }

   public synchronized void writeStart() {
        writersWaiting++;
        while (writing || readers>0)
           try { wait(); }
             catch(Exception e) { };
        writersWaiting--;
        writing=true;
    }
    public synchronized void writeEnd() {
        writing=false;
        notifyAll();
    }
}
```

8.1 Describa brevemente qué realiza este código.

8.2 Escriba un ejemplo de código que permita implementar tanto a los hilos lectores como a los hilos escritores, ambos haciendo uso del mismo objeto compartido y utilizando los métodos de la clase *ReadersWriters* para el control del acceso a dicho objeto.

8.3 Indique si el código de la clase *ReadersWriters* permite controlar de forma correcta el acceso al recurso compartido. Es decir, un acceso concurrente de los hilos lectores y un acceso en exclusión mutua de los hilos escritores. Puede mostrar alguna traza que justifique su respuesta.

8.4 Indique las modificaciones a realizar en la clase *ReadersWriters* para dar prioridad a los hilos lectores, de modo que si hay hilos escritores e hilos lectores que quieren hacer uso del recurso compartido, accederán primero los lectores. Si no fuera necesario realizar ninguna modificación, indíquelo.

SOLUCIÓN DE LAS ACTIVIDADES

A continuación se proporciona la solución de las actividades anteriores. Para muchas de ellas, se remitirá a las trasparencias del tema donde estén descritos los conceptos por los que se pregunta, así como a las secciones y/o capítulos correspondientes del libro de referencia de la asignatura:

> Francisco Muñoz, Estefanía Argente, Agustín Espinosa, Pablo Galdámez, Ana García-Fornes, Rubén de Juan, Juan Salvador Sendra. *Concurrencia y Sistemas Distribuidos.* Editorial Universidad Politécnica de València, ISBN: 978-84-8363-986-3, Ref. editorial 599, 1a Edición 2012.

ACTIVIDAD 1.

1.1 Vea Unidad 1, sección "Ventajas e inconvenientes de la programación concurrente" (trasparencias 6 y 7). Vea también en el libro, sección *1.3.1 Ventajas e inconvenientes*.

1.2 Vea Unidad 2, sección "Determinismo", trasparencia 26. Vea también en el libro, sección *2.5 Determinismo*.

ACTIVIDAD 2.

2.1 Vea Unidad 2, sección Cooperación entre hilos, a) Comunicación (trasparencia 12). Vea también en el libro, sección *2.3.1 Comunicación*.

2.2 La cooperación o colaboración entre actividades requiere tanto de un mecanismo de comunicación, como de mecanismos de sincronización. Por ejemplo, si en la comunicación se utiliza memoria compartida, al menos se requerirá hacer uso del mecanismo de sincronización de exclusión mutua para garantizar que no se produzcan condiciones de carrera. Vea también en el libro, sección *2.3 Cooperación entre hilos.*

2.3 Vea Unidad 2, sección "Modelo de Ejecución", trasparencias 16 y 17. Vea también en el libro, sección *2.4 Modelo de Ejecución*.

2.4 Vea Unidad 2, sección "Sección crítica", trasparencias 28-31. Vea también en el libro, sección *2.6 Sección Crítica*.

ACTIVIDAD 3.

3.1 Vea Unidad 2, sección "Ciclo de vida de los hilos Java", trasparencias 5-7. Vea también en el libro, sección *2.2.1 Ciclo de vida de los hilos Java.*

3.2 Vea Unidad 2, sección "Ciclo de vida de los hilos Java", trasparencia 8. Vea también en el libro, sección *2.2.1 Ciclo de vida de los hilos Java.*

3.3 a) El ejemplo mostrado se corresponde con la definición de un hilo usando una clase anónima que extiende a la clase Thread. El hilo se crea y se lanza directamente a ejecución. La clase anónima se ha creado en el método main de la clase NewThread.

b) Ejemplos similares del resto de clasificaciones serían:

b1) Definición de un hilo usando una clase con nombre y extendiendo de Thread:

```
public class MyThread extends Thread{
     protected String name;
     public MyThread(String n){name=n;}

      public void run() {
           System.out.println("Esto es un hilo.");
      }
}

public class NewThread{
     public static void main(){
          MyThread t=new MyThread("Hilo");
          t.start();
     }
}
```

b2) Definición de un hilo usando una clase con nombre e implementando la interfaz Runnable:

```
 public class MyThread implements Runnable{
      protected String name;
      public MyThread(String n){name=n;}

      public void run() {
             System.out.println("Esto es un hilo."); }
         };
  }

 public class NewThread{

      public static void main(){

         Thread t=new Thread(new MyThread("Hilo"));

         t.start();

      }

}
```

b3) Definición de un hilo usando una clase anónima e implementando la interfaz Runnable:

```
public class NewThread{

    public static void main(){

        new Thread(new Runnable() {

            public void run() {

            System.out.println("Esto es un hilo.");

            }

         }).start();

} }
```

ACTIVIDAD 4

4.1 La clase *RaceCondition* contiene el método *main*, donde se crea un objeto *c* de tipo *Counter* y dos hilos (de tipo *MyThread*), que se lanzan ejecución a continuación (con *tA.start()* y *tB.start()*). A cada uno de esos hilos, en el momento de su creación, se les pasa el objeto *c* (que será compartido por ellos), un identificador (A ó B) y un número determinado.

La clase *MyThread* (que extiende de la clase Thread), recibe en un su constructor la referencia a un objeto Counter, así como un nombre y un número (de tipo long). Cualquier hilo de esta clase, al ejecutarse, hará uso del objeto Counter, incrementando su valor con el método c.add(i). A continuación, usando el método Thread.yield() cede su tiempo de CPU (pasando así al estado ready-to-run) y, cuando vuelva a ser seleccionado por el planificador, imprime un mensaje con su nombre y el valor que ha añadido a Counter y se duerme un intervalo de tiempo. Al despertar, consulta el valor que tiene Counter (con el método c.getCount()) y lo imprime en pantalla.

Finalmente, la clase Counter representa a un objeto con un contador, cuyo valor inicial es 0, que contiene dos métodos: *add*, para añadir un valor determinado al contador; y *getCount*, que permite consultar dicho valor. En el método *add*(), inicialmente se fuerza al hilo en ejecución a ceder su tiempo de CPU. Además, el incremento del valor del contador se realiza en varios pasos, haciendo uso de una variable interna "value".

Por tanto, este código lanza dos hilos en ejecución que, de forma concurrente, harán uso del mismo objeto compartido *c*, de tipo Counter, incrementando su valor y leyendo posteriormente el estado de dicho objeto.

4.2 Este código **no es determinista**, pues tenemos dos hilos ejecutándose concurrentemente, que hacen uso de un mismo objeto compartido y, además, sin ningún tipo de protección de acceso sobre el objeto, es decir, sin ningún método de sincronización.

Ejemplos de ejecución de este código son (cada ejemplo se muestra en una columna distinta):

----------	----------	----------	----------
Thread #B adds: 2	Thread #A adds: 3	Thread #B adds: 2	Thread #A adds: 3
Thread #A adds: 3	Thread #B adds: 2	Thread #A adds: 3	Thread #B adds: 2
Thread #B reads: 3	Thread #B reads: 2	Thread #B reads: 3	Thread #B reads: 5
Thread #A reads: 3	Thread #A reads: 2	Thread #A reads: 3	Thread #A reads: 5

Como se ve, el resultado final del objeto c, de tipo Counter, (mostrado en última línea de cada ejecución) unas veces es 3, otras 2 y otras 5. Por tanto, para las mismas entradas se obtienen resultados distintos, por lo que el código no es determinista.

4.3 Efectivamente, se pueden producir condiciones de carrera, al tener dos hilos que hacen uso del mismo objeto compartido (objeto c de tipo Counter), modificando y leyendo su estado a la vez. En el ejemplo, si el valor inicial del objeto c es 0 y tenemos dos hilos que incrementan su valor, uno de ellos en 2 unidades y el otro en 3, se esperaría que el valor final del objeto fuera 5. Sin embargo, en las trazas mostradas anteriormente podemos ver que en algunos casos el resultado final es 3, o bien es 2. Estos resultados son ejemplos de que se han producido condiciones de carrera.

4.4 En este ejemplo, el acceso al objeto de tipo Counter es una sección crítica, tanto en los métodos que actualizan su estado como en los que lo consultan. En Java, se protegen añadiendo la etiqueta "synchronized" a los métodos add y getCount. Quedaría como sigue:

```
public synchronized void add(long x){
....
}
public synchronized long getCount(){
   return count;
}
```

ACTIVIDAD 5

5.1 a) En este ejemplo se requiere hacer uso de la "exclusión mutua" para proteger el acceso al objeto *Counter*, ya que se lanzan varios hilos que, de forma concurrente, hacen uso de los métodos *increment*(), *decrement*() y *value*() del mismo objeto *Counter* compartido. Por tanto, se tienen varios hilos que, en paralelo, incrementan, decrementan o bien consultan el valor del objeto *Counter*. Para evitar las condiciones de carrera que puedan aparecer, el acceso a esos métodos debe realizarse en exclusión mutua, de modo que mientras un hilo esté actualizando (o consultando) el valor del objeto, otro hilo no pueda acceder a dicho objeto.

b) En este ejemplo no se requiere de sincronización condicional, entendida ésta como la obligación a que ciertos hilos se suspendan mientras no se cumpla una determinada condición.

c) Sí que se requiere cierta sincronización entre los hilos, ya que el hilo principal debe esperarse a que acaben el resto de hilos (es decir, los hilos incrementadores y los decrementadores) antes de leer el valor del objeto c, con el método *c.value*().

5.2 Para garantizar la exclusión mutua, basta con añadir la etiqueta "synchronized" a todos los métodos que modifiquen y/o lean el atributo "c" de la clase Counter. En concreto:

```
class Counter{
   private int c=0;
   public synchronized void increment(){ c++; }
   public synchronized void decrement(){c--;}
   public synchronized int value(){return c;}
}
```

Para la sincronización entre el hilo principal y los hilos incrementadores/decrementadores, en este caso se requiere que el hilo principal imprima el valor final del contador tras la ejecución de los otros cuatro hilos. Dicha espera se implementará con el método *join().*

```
public class RaceCondition {
  .....
   inc1.start(); inc2.start();
   dec1.start(); dec2.start();
   try{
       inc1.join();    inc2.join();
       dec1.join();    dec2.join();
   } catch (InterruptedException e){};
   System.out.println("Final result "+c.value() );
}
```

Nota: En este ejemplo concreto, como el hilo *main* es el único que hace uso del método *value*(), justo después de que hayan acabado el resto de hilos, este código también funcionaría de forma correcta si no se le pusiera la etiqueta "synchronized" en el método *value*().

Sin embargo, resulta conveniente implementar siempre clases con "código thread-safe", es decir, con código que pueda ser ejecutado concurrentemente por distintos hilos de forma segura. Por tanto, el método *value*() debería estar también protegido con la etiqueta "synchronized", ya que como diseñadores no podemos conocer siempre de antemano el modo en el que los hilos van a hacer uso de los métodos de los objetos que diseñemos. En este caso, el método *value*() podría haber sido también utilizado por los hilos incrementadores y/o decrementadores.

ACTIVIDAD 6

6.1 En este programa se lanzan un hilo consumidor y un hilo productor que actúan sobre el mismo buffer, de tamaño 1 elemento, inicialmente vacío. El productor inserta un elemento en el buffer de forma repetitiva (100 veces), mientras que el consumidor recoge un elemento del buffer de forma repetitiva (100 veces). Como los métodos de acceso al buffer compartido no proporcionan ningún mecanismo de sincronización, el acceso concurrente al mismo puede producir condiciones de carrera (por ejemplo, que el consumidor lea del buffer estando éste vacío). Por tanto, este programa no es determinista, ya que tras distintas ejecuciones podemos obtener distintos valores almacenados en el buffer.

6.2 Como hemos dicho antes, sí que se pueden producir condiciones de carrera. Por ejemplo, podríamos tener la siguiente traza:

```
Producer #2 puts 1
Consumer #1 gets 1
Producer #2 puts 2
Producer #2 puts 3
Consumer #1 gets 3
```

Y si lo que se quería era que el consumidor obtuviera todos y cada uno de los valores que el hilo productor deja en el buffer, entonces en este caso el valor "2" se habría perdido.

6.3 Se requiere implementar exclusión mutua para el acceso a los métodos del objeto Buffer. Se requiere implementar sincronización condicional en los métodos *put*() y *get*() del objeto Buffer, para así controlar el acceso al buffer cuando está lleno y cuando está vacío.

La tabla a rellenar quedaría como sigue:

Método	Espera cuando	Avisa a
put()	buffer lleno	Thread que espera porque el buffer estaba vacío
get()	buffer vacío	Thread que espera porque el buffer estaba lleno

6.4 Los cambios realizados en el código para implantar tanto la exclusión mutua como la sincronización condicional se muestran en negrita en el código siguiente:

```
1.1   public class Buffer {
1.2     private int store = 0;
1.3     private boolean full = false;
1.4
1.5     public synchronized int get()  {
            while (!full) try {
                wait();
            }  catch (InterruptedException e) {};
1.6         int value = store;
1.7         store = 0;
1.8         full = false;
            notifyAll();
1.9         return value;
1.10    }
1.11
1.12    public synchronized void put(int value)   {
            while (!full) try {
                wait();
            }  catch (InterruptedException e) {};
1.13        full = true;
1.14        store = value;
1.15    }
1.16  }

2.1   public class Main {
2.2     public static void main(String[] args)  {
2.3         Buffer c = new Buffer();
2.4         Consumer c1 = new Consumer(c, 1);
2.5         Producer p1 = new Producer(c, 2);
2.6         c1.start();
2.7         p1.start();
2.8         System.out.println("Producer and " +
2.9           "Consumer have terminated.");
2.10    }
2.11  }
```

```
3.1   public class Consumer extends Thread {
3.2     private Buffer b;
3.3     private int number;
3.4     public Consumer(Buffer ca, int id)  {
3.5       b = ca;
3.6       number = id;
3.7     }
3.8     public void run()  {
3.9       int value = 0;
3.10      for (int i = 1; i < 101; i++)
3.11      {
3.12        value = b.get();
3.13        System.out.println("Consumer #" + number +
3.14        " gets: " + value);
3.15      }
3.16    }
3.17  }

4.1   public class Producer extends Thread {
4.2     private Buffer b;
4.3     private int number;
4.4     public Producer(Buffer ca, int id) {
4.5       b = ca;
4.6       number = id;
4.7     }
4.8     public void run()  {
4.9       for (int i = 1; i < 101; i++)
4.10      {
4.11        b.put(i);
4.12        System.out.println("Producer #" + number +
4.13        " puts: " + i);
4.14      }
4.15    }
4.16  }
```

Como se observa, solamente se requiere actualizar la clase Buffer, que actuará ahora como un monitor.

6.5 En el caso de la sincronización entre el hilo principal y los demás hilos, se puede implementar utilizando el método join() de la clase Thread. Se muestra a continuación el código resultante (sobre la solución del ejercicio anterior), remarcando en negrita los nuevos cambios:

```
1.1   public class Buffer {
1.2     private int store = 0;
1.3     private boolean full = false;
1.4
1.5     public synchronized int get()   {
            while (!full) try {
                wait();
            }  catch (InterruptedException e) {};
1.6         int value = store;
1.7         store = 0;
1.8         full = false;
            notifyAll();
1.9         return value;
1.10    }
1.11
1.12    public synchronized void put(int value)    {
            while (!full) try {
                wait();
            }  catch (InterruptedException e) {};
1.13        full = true;
1.14        store = value;
1.15    }
1.16  }

2.1   public class Main {
2.2     public static void main(String[] args)  {
2.3         Buffer c = new Buffer();
2.4         Consumer c1 = new Consumer(c, 1);
2.5         Producer p1 = new Producer(c, 2);
2.6         c1.start();
2.7         p1.start();
            try {
                c1.join();  p1.join();
            }  catch (InterruptedException e) {};
2.8         System.out.println("Producer and " +
2.9           "Consumer have terminated.");
2.10    }
2.11  }
```

```
3.1   public class Consumer extends Thread {
3.2     private Buffer b;
3.3     private int number;
3.4     public Consumer(Buffer ca, int id)  {
3.5       b = ca;
3.6       number = id;
3.7     }
3.8     public void run()  {
3.9       int value = 0;
3.10      for (int i = 1; i < 101; i++)
3.11      {
3.12        value = b.get();
3.13        System.out.println("Consumer #" + number +
3.14        " gets: " + value);
3.15      }
3.16    }
3.17  }

4.1   public class Producer extends Thread {
4.2     private Buffer b;
4.3     private int number;
4.4     public Producer(Buffer ca, int id) {
4.5       b = ca;
4.6       number = id;
4.7     }
4.8     public void run()  {
4.9       for (int i = 1; i < 101; i++)
4.10      {
4.11        b.put(i);
4.12        System.out.println("Producer #" + number +
4.13        " puts: " + i);
4.14      }
4.15    }
4.16  }
```

ACTIVIDAD 7

7.1 Según el concepto de monitor, cuando el hilo que está dentro del monitor ejecuta un notify() sobre una condición, se reactiva a algún hilo que se hubiera suspendido en dicha condición. Sin embargo, solamente uno de esos dos hilos (el que está ya en el monitor y el que acaba de ser despertado) puede continuar activo en el monitor. Las variantes de monitor presentan precisamente distintas alternativas, en función de que continué su ejecución uno u otro hilo.

Para más detalles, vea la Unidad 3, sección Monitor – Variantes, trasparencia 23, o bien en el libro de referencia, sección *3.3 Variantes*.

7.2 Las características que cumplen las variantes de monitor son las siguientes:

- Modelo de Brinch Hansen:
  - El modelo obliga a que toda invocación a notify() sea la última sentencia en los métodos del monitor en los que aparezca tal invocación.
  - El modelo jamás suspende al hilo que invoque a notify().

- Modelo de Hoare:
  - El modelo necesita una cola especial de entrada
  - En caso de que haya hilos suspendidos en alguna condición, suspende al hilo que ha invocado a notify(), activando a uno de los hilos que llamó antes a wait().
  - El modelo suspende (y deja en una cola especial de entrada) al hilo que ha invocado a notify(), activando a uno de los hilos que llamó antes a wait().

- Modelo de Lampson & Redell:
  - El modelo jamás suspende al hilo que invoque a notify().

  Nota: en el modelo de Lampson & Redell, el hilo reactivado pasa directamente a la cola de entrada. No hay una cola especial de entrada distinta, sino que se emplea una única cola, donde esperan tanto los hilos reactivados como los que lleguen nuevos al monitor. Cuando el hilo consiga entrar en el monitor, el estado por el que estaba esperando puede haber cambiado de nuevo, por lo que tendrá que reevaluar la condición.

ACTIVIDAD 8

8.1 El código mostrado implementa la clase ReadersWriters que, en su inicialización, establece la cantidad de lectores a 0 (readers=0), la cantidad de escritores en espera a 0 (writersWaiting=0), así como que el estado del recurso es que "no se está escribiendo" (writing=false). Esta clase ofrece también los métodos siguientes:

- readStart(): si se está escribiendo o bien hay escritores esperando, el hilo que lo ejecuta (i.e. un lector) tendrá que esperar. En caso contrario, incrementa en uno el valor de la variable *readers* (i.e. hay un lector más accediendo al recurso).
- readEnd(): se indica que hay un lector menos accediendo al recurso y se notifica a todos los hilos en espera (en el monitor Java).
- writeStart(): se indica que hay un escritor más en espera. Si hay otro hilo escribiendo o hay lectores accediendo al recurso (condición writting || readers >0) entonces se queda a la espera. En caso contrario, se indica que hay un escritor menos en espera y que se está accediendo al recurso para escritura.
- writeEnd(): se indica que ya no se accede al recurso para escritura (writting=false) y se notifica a todos los hilos en espera (en el monitor Java).

8.2 A continuación se muestran las clases ***Reader***, ***Writer*** y ***RWExample***, con el código necesario para implementar a los hilos lectores, escritores y el hilo principal, respectivamente.

```
public class Reader extends Thread {
    private int id;
    private int cycles;
    private ReadersWriters controller;

    public Reader(int id, int loops, ReadersWriters rws) {
        this.id = id;
        this.cycles=loops;
        this.controller=rws;
    }
   public void run() {
        int i = 0;
        while (i < cycles) {
            controller.readStart();
            // Leyendo del recurso. Ejemplo:
            // resource.readerReading();
            // Esto lleva tiempo.Se puede simular con un sleep()
            controller.readEnd();
         i++;
        }
   }
}


public class Writer extends Thread {
    private int id;
    private int cycles;
    private ReadersWriters controller;

    public Writer(int id, int loops, ReadersWriters rws) {
        this.id = id;
        this.cycles=loops;
        this.controller=rws;
    }
   public void run() {
        int i = 0;
        while (i < cycles) {
            controller.writeStart();
            // Leyendo del recurso. Ejemplo:
            // resource.writerWriting();
            // Esto lleva tiempo.Se puede simular con un sleep()
            controller.writeEnd();
         i++;
        }
   }
}
```

```
public class RWExample {
    static public void main(String[] args) {

      ReadersWriters controllerRW = new ReadersWriters();
      //Se crean a continuación varios lectores
      //y varios escritores
      int nReaders = 2;
      int nWriters = 2;
      int loops=100;

      Reader readers[] = new Reader[nReaders];
      Writer writers[] = new Writer[nWriters];

      int i = 0;
      for (Reader n : readers) {
             n = new Reader(i, loops, controllerRW);
             i++;
             n.start();
         }

      i = 0;
      for (Writer e : writers) {
             e = new Writer(i, loops, controllerRW);
             i++;
             e.start();
         }
   }
}
```

8.3 Si dos hilos lectores quieren acceder al recurso compartido concurrentemente, ambos ejecutarán el método *readStart*(). Este método de la clase ReadersWriters lleva la etiqueta synchronized, de modo que solamente uno de los dos hilos lectores podrá ejecutarlo inicialmente y, cuando el hilo termine el método o bien se suspenda con un wait(), el otro hilo lector podrá ejecutarlo.

Dentro del método, como se ha indicado en la respuesta al ejercicio 8.1, si hay algún otro hilo escribiendo en el recurso o bien hay escritores esperando, el lector tendrá que esperar. Por tanto, si ya hay un hilo escritor usando el recurso, el lector tendrá que esperar, de modo que se garantiza la exclusión mutua del hilo escritor. Si no hay ningún hilo escritor escribiendo o esperando, el lector podrá acceder al recurso. Cuando acceda, como habrá salido del método readStart(), el otro hilo lector ya podrá ejecutar dicho método y, si no hay ningún escritor en espera, podrá también acceder al recurso. Por tanto, tendríamos dos hilos lectores accediendo al recurso de forma concurrente.

Cuando estos hilos terminen su acceso al recurso, actualizarán (en el método readEnd) la variable readers, indicando que hay un lector menos, y reactivarán a todos los hilos suspendidos en el monitor Java. Todos los métodos wait() que se han utilizado en la clase ReadersWriters están dentro de bucles while, los hilos que se reactiven volverán a evaluar sus condiciones, suspendiéndose si no se cumple todavía la condición por la que esperaban.

Además, las notificaciones (notifyAll) se realizan siempre justo antes de salir del monitor, por lo que no se producen variaciones en el estado que acaba de notificarse.

Por tanto, el comportamiento de la clase ReadersWriters respecto al control del acceso de los hilos lectores es correcto.

Para los hilos escritores, se puede observar que, de forma análoga, se realiza también un control correcto del acceso al recurso. En este caso, si dos hilos escritores desean acceder de forma concurrente (con el método writeStart), uno de ellos conseguirá el acceso al monitor y podrá ejecutar el método writeStart. En dicho método indicará primero que desea acceder (incrementando la variable writersWaiting) y luego comprobará si se está escribiendo o leyendo en el recurso. Si es así, deberá esperar en la condición del monitor. Si el recurso está libre, decrementa la variable writersWaiting y pone a cierto la variable writing, para indicar que hace uso del recurso en modo de escritura. Al terminar el método writeStart(), saldrá del monitor.

Cuando el otro hilo escritor ejecute el método writeStart, al comprobar su condición (writing || readers>0) se quedará a la espera en la condición interna (o implícita) del monitor Java, por lo que se garantiza el acceso en exclusión mutua de los hilos escritores.

Por tanto, el comportamiento de la clase ReadersWriters respecto al control del acceso de los hilos escritores también es correcto.

8.4 Las modificaciones a realizar para dar prioridad a los hilos lectores se muestran en negrita en el siguiente código:

```
public class ReadersWriters {
   private int writersWaiting;
   private boolean writing;
   private int readers;

   public ReadersWriters() {
       writersWaiting=readers=0;
       readersWaiting=0;
       writing=false;
   }

   public synchronized void readStart()
     {
    readersWaiting++;
    while (writing)
          try { wait(); }
            catch(Exception e) { };
    readersWaiting--;
    readers++;
   }
   public synchronized void readEnd() {
       readers--;
       notifyAll();
   }

   public synchronized void writeStart()
{
       writersWaiting++;
       while (writing || readers>0 ||
          readersWaiting)
          try { wait(); }
            catch(Exception e) { };
       writersWaiting--;
       writing=true;
   }
   public synchronized void writeEnd() {
       writing=false;
       notifyAll();
   }
}
```

Nota: la variable *writersWaiting* y sus actualizaciones podrían eliminarse del código, pues en este caso no se utilizan para las condiciones.

Nota: En el método *writeStart* se podría haber mantenido la misma condición anterior, es decir, `while (writing || readers>0)`, ya que si hay lectores esperando será debido a que hay algún escritor usando el recurso, es decir, `writing=true`.