# Programación Concurrente y de Tiempo Real

## Guión de prácticas 7: Control de la Concurrencia en Java (API estándar)

Natalia Partera Jaime
Alumna colaboradora de la asignatura

# Índice

# 1. Introducción

Ya hemos visto que, en ocasiones, varias operaciones pueden intentar ejecutarse concurrentemente sobre un mismo dato, cada una ejecutándose desde un hilo distinto. Sabemos, además, que para mantener la consistencia y el buen funcionamiento, debemos controlar el acceso de estas operaciones sobre los datos mediante algún mecanismo de control, como la exclusión mutua.

En Java, la unidad mínima de datos son los objetos. Es por esto que en Java la exclusión mutua se controla a nivel de objetos. Cada objeto tiene asociado un cerrojo (o *lock*) que permite, o no, que un hilo dado tenga acceso al objeto. El cerrojo de un objeto impide que otros hilos puedan utilizar el objeto si está siendo utilizado por un hilo concreto. Cuando este hilo termine de utilizar el objeto y quede liberado, el cerrojo se abrirá y permitirá que uno de los hilos que se encontraban a la espera pueda utilizar el objeto, volviendo a cerrarse mientras este hilo no haya acabado. Estos cerrojos son mutuamente exclusivos, lo cual quiere decir que sólo un hilo puede controlar el cerrojo a la vez y un hilo sólo puede controlar un cerrojo al mismo tiempo.

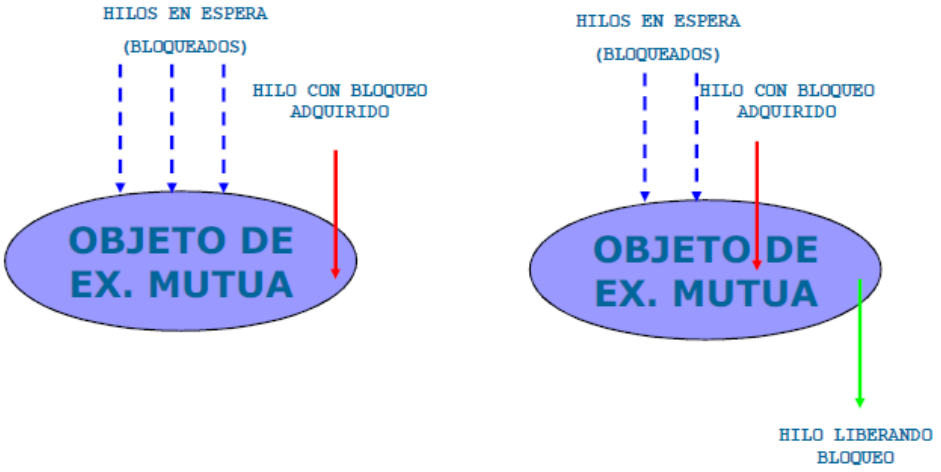


Figura 1: Imagen que representa el funcionamiento de un cerrojo.

Existen varios mecanismos para implementar la concurrencia en Java. Por ejemplo, se puede utilizar la etiqueta `synchronized` que provee Java para sincronizar código y conseguir exclusión mutua. También se pueden usar otros mecanismos como los monitores o los semáforos.

# 2. Exclusión mutua entre hilos

Unas de las formas de lograr exclusión mutua en Java es usando la etiqueta `synchronized`. Esta etiqueta es proporcionada por el propio lenguaje y otorga propiedades especiales al código que la utilice: sólo un fragmento de código con esta etiqueta puede ser ejecutado a la vez.

Para lograr la exclusión mutua entre dos o más hilos en Java con la etiqueta `synchronized` se pueden utilizar las siguientes técnicas de control:

- **Bloques sincronizados:** El fragmento de código que haya que ejecutar bajo exclusión mutua se incluye en un bloque etiquetado como `synchronized`. Actúa como una región crítica.
- **Métodos sincronizados:** Los objetos que deban manejarse bajo exclusión mutua se encapsulan en una clase y todos sus métodos modificadores se etiquetan como `synchronized`.

## 2.1. Exclusión mutua entre hilos: bloques de código sincronizado

Los bloques de código sincronizado actúan como si se tratase de una región crítica. La sintáxis de estos bloques es la que sigue:

```
public metodo (tipo1 arg1, tipoN argN) {
  /*Código que no se ejecuta en Exclusión Mutua, si hay.*/

  /*Inicio de la región crítica.*/
  synchronized(objetoS) {
    /*Código en Exclusión Mutua.*/

  }
  /*Fin de la región crítica.*/

  /*Código que no se ejecuta en Exclusión Mutua, si hay.*/
}
```

La ejecución de un bloque de código sincronizado depende del cerrojo de un objeto concreto. Cómo puede observar en el código anterior, el bloque `synchronized` tiene como argumento un objeto llamado `objetoS`. Cuando el objeto `objetoS` no está siendo utilizado y el hilo en el que se ejecuta el método `metodo` llega al bloque `synchronized`, el bloque `synchronized` recibe el cerrojo del objeto `objetoS`. En este caso, el bloque `synchronized` se ejecutará, cambiando el estado del cerrojo del objeto `objetoS` durante la ejecución del bloque. Si el objeto estuviera siendo usado, el hilo se queda esperando hasta que el objeto se librere y el bloque `synchronized` pueda tomar el control sobre el cerrojo del objeto.

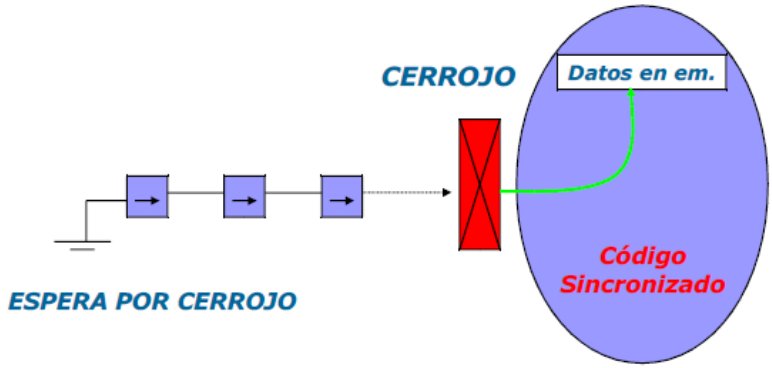


Figura 2: Imagen que muesta el comportamiento de hilos en espera de un cerrojo.

Cualquier objeto puede ser pasado como argumento de un bloque `synchronized`, incluso el propio objeto desde el que se ejecuta el bloque. Siempre que sea posible, el bloque tomará el control del cerrojo del objeto y ejecutará su código.

Veamos un ejemplo sencillo sobre cómo se maneja un bloque de código sincronizado. Supongamos una clase con una variable de la que queremos hacer una cuenta atrás y un programa que realice la cuenta atrás con varios hilos. La clase necesitará, al menos, la variable y un método para cambiar su valor. Un ejemplo para ello sería la siguiente clase:

```
/**
 * Clase CuentaAtras.
 *
 * @author Natalia Partera
 * @version 1.0
 */

public class CuentaAtras {

  //Atributo privado
  private int cont;

  //Constructor de la clase
  public CuentaAtras(int num) {
    if(num > 0)
      cont = num;
    else
      cont = 0;
  }

  //Método modificador que decrementa la variable
  public void contar() {
    System.out.println("Actualizando...");
    synchronized(this) {
      if(cont > 0) {
        System.out.println("Contador actualizado: " + cont);
        --cont;
      }
    }
    System.out.println("Variable actualizada.");
  }

  //Método observador que devuelve el valor de la variable
  public int valor() {
    synchronized(this) {
      return cont;
    }
  }

}
```

El programa que realice la cuenta atrás contendrá un objeto de la clase anterior y lanzará varios hilos para acelerar la cuenta atrás. El siguiente código muestra un programa con estas características:

```
/**
 * Programa que realiza una cuenta atrás usando varios hilos.
 *
```

```
 * @author Natalia Partera
 * @version 1.0
 */

public class UsaCuentaAtras extends Thread {

  //Variable compartida que debe ser ejecutada en EM.
  private static CuentaAtras cont;

  //Código que ejecuta cada hilo
  public void run() {
    while(cont.valor() > 0) {
      cont.contar();
    }
  }

  //Programa principal
  public static void main (String[] args) {
    cont = new CuentaAtras(35);
    UsaCuentaAtras hilo1 = new UsaCuentaAtras();
    UsaCuentaAtras hilo2 = new UsaCuentaAtras();
    UsaCuentaAtras hilo3 = new UsaCuentaAtras();

    hilo1.start();
    hilo2.start();
    hilo3.start();
  }

}
```

---

**Ejercicio 1** Pruebe el programa anterior. ¿Funciona como esperaba? ¿Son imprescindibles los dos bloques `synchronized`? ¿Qué pasaría si eliminase el bloque `synchronized` del método `valor()` de la clase `CuentaAtras`? ¿Y si elimina también el bloque `synchronized` del método `contar()`? Realice los cambios y compruebe si estaba en lo cierto.

---

**Ejercicio 2** Aumente la cuenta atrás, por ejemplo a 50 y compruebe si tras cada decremento aparece el mensaje `Variable actualizada.` ¿A qué se debe? Modifique el programa anterior para que el mensaje `Variable actualizada.` aparezca siempre justo después de decrementar el contador. Compruebe sus cambios con los que se explican en el apartado 5.1.

---

El uso de bloques sincronizados para controlar la exclusión mutua resulta útil cuando es necesario adaptar un trozo de código secuencial a un entorno concurrente. También se utiliza cuando es necesario sincronizar sólo algunas líneas de código de un método.

Para controlar la exclusión mutua con bloques sincronizados hay que identificar en qué parte del código se accede al recurso compartido y controlar el acceso. Para la utilización de bloques sincronizados necesitaremos definir un objeto que sirva para controlar la exclusión mutua, o podemos usar el propio

objeto (`this`). Por último, sincronizaremos el código crítico introduciéndolo en un bloque `synchronized` y utilizando el objeto de control que mencionábamos antes para el cerrojo.

---

**Ejercicio 3** Simule un sistema de expedición de citas. En él, múltiples usuarios pueden conectarse al sistema para solicitar una cita, consultar las citas disponibles o cancelar su cita. El sistema tendrá un número limitado de citas para cada día. Lance varios hilos que simulen ser los clientes y solicite citas hasta agotar las citas disponibles para el día dado. Cancele también alguna cita y vuelva a solicitarla. Realice este ejercicio utilizando bloques sincronizados. Puede comprobar su solución con la que encontrará en el apartado 5.2.

---

## 2.2. Exclusión mutua entre hilos: métodos sincronizados

Los métodos sincronizados funcionan como si fuesen operaciones atómicas, dado que durante su ejecución no pueden ser interrumpidos. A continuación se muestra la estructura que siguen estos métodos:

```
public synchronized metodo (tipo1 arg1, tipoN argN) {
  /*Código en Exclusión Mutua.*/

}
```

Cuando un método etiquetado como `synchronized` está siendo ejecutado, no se permite la ejecución de otro método `synchronized` del mismo objeto ni otra invocación del método en ejecución (ya que supone invocar al mismo objeto en ejecución usando el mismo método) hasta que el método `synchronized` no termine su ejecución. En cambio, mientras que un método `synchronized` está siendo ejecutado, pueden ejecutarse otros métodos no sincronizados.

Esto se debe, en realidad, a que un método sincronizado toma el control del cerrojo del objeto que lo invoca. De este modo, este método no puede ser invocado desde otros hilos, porque ese método ya está siendo ejecutado y controla un cerrojo. Así mismo, otros métodos sincronizados del mismo objeto no pueden ser ejecutados porque el control sobre el cerrojo de ese objeto común ya lo tiene el otro método.

Veamos con un ejemplo cuando se ejecutan o no métodos sincronizados. Supongamos que tenemos una clase que representa un teléfono y que puede realizar o recibir llamadas, pero no ambas a la vez:

```
/**
 * Clase Telefono.
 *
 * @author Natalia Partera
 * @version 1.0
 */

public class Telefono implements Runnable{

  //Atributos privados
  private int id;

  //Constructor
  public Telefono() {}
  public Telefono(int id) {
```

```
    this.id = id;
  }

  //Métodos
  public synchronized void LlamadaEntrante() {
    System.out.println("Teléfono " + id + ": Iniciando llamada entrante.");
    for(int i = 0; i < 500; ++i) {}
    System.out.println("Teléfono " + id + ": Llamada entrante en curso.");
    for(int i = 501; i < 1000; ++i) {}
    System.out.println("Teléfono " + id + ": Finalizando llamada entrante.");
  }

  public synchronized void LlamadaSaliente() {
    System.out.println("Teléfono " + id + ": Iniciando llamada saliente.");
    for(int i = 0; i < 500; ++i) {}
    System.out.println("Teléfono " + id + ": Llamada saliente en curso.");
    for(int i = 501; i < 1000; ++i) {}
    System.out.println("Teléfono " + id + ": Finalizando llamada saliente.");
  }

  //Método run()
  public void run() {
    for(int i = 0; i < 5; ++i) {
      LlamadaEntrante();
      LlamadaSaliente();
    }
  }

}
```

Supongamos ahora una centralita donde hay varios teléfonos. Se ejecutan varios hilos que representan a distintos teléfonos. Cada teléfono podrá recibir o realizar una llamada a la vez, aunque varios teléfonos pueden estar ocupados al mismo tiempo.

```
/**
 * Programa que simula el comportamiento de una centralita con varios teléfonos.
 *
 * @author Natalia Partera
 * @version 1.0
 */

public class Centralita {

  //Programa principal
  public static void main (String[] args) {
    Thread tel1 = new Thread(new Telefono(1));
    Thread tel2 = new Thread(new Telefono(2));
    Thread tel3 = new Thread(new Telefono(3));

    tel1.start();
    tel2.start();
    tel3.start();
```

```
    }

}
```

Al ejecutar este ejemplo, podemos comprobar que los pasos que se siguen al ejecutar un método sincronizado son:

- Si un teléfono dado está realizando una llamada (por ejemplo, también valdría si está recibiendo una llamada) y el teléfono recibe una llamada (o intenta realizarla si estamos en el ejemplo contrario), ésta llamada queda en espera. Esto se produce porque ambos métodos (`LlamadaEntrante()` y `LlamadaSaliente()` son sincronizados) y mientras que se está llevando a cabo uno de los dos en un objeto, no puede darse otro método sincronizado sobre ese objeto.
- Cuando la llamada que estaba en espera pasa a ser ejecutada en el teléfono, el teléfono que la gestiona es bloqueado, por lo que no puede recibir ni enviar otras llamadas a la vez.
- La llamada se lleva a cabo (se ejecuta el método).
- Y por último, una vez que la llamada termina, se desbloquea el teléfono, por lo que puede volver a recibir o enviar llamadas. De este modo, si durante el tiempo de ejecución de la llamada anterior, se produjo alguna nueva y se quedó en espera, ésta es desbloqueada también.

Otras consideraciones a tener en cuenta son las siguientes:

- Los métodos ya etiquetados como `static` también pueden ser etiquetados como `synchronized`.
- La condición de método sincronizado es válida para la clase en la que se define. Si una clase hereda de otra, los métodos de la subclase pueden ser sincronizados o no independientemente de cómo fueran en la superclase.

Para controlar la exclusión mutua en Java utilizando métodos sincronizados hay que encapsular el recurso crítico en una clase y etiquetar, al menos, sus métodos modificadores como `synchronized`. A continuación, se crean varios hilos que compartan un objeto de la clase anteriormente creada y se lanzan.

Hay que tener en cuenta, además, que un método sincronizado puede invoca a otro método sincronizado sobre el mismo objeto. También se permiten llamadas recursivas a métodos sincronizados.

---

**Ejercicio 4** Cree un programa que simule el comportamiento de un reproductor de música. El reproductor debe almacenar en todo momento el número de la pista que está reproduciendo. También debe simular el comportamiento de sus botones: *Reproducir*, *Pausar*, *Parar*, *Anterior* y *Siguiente*. Cada botón debe mostrar un mensaje cuando es pulsado. Puede hacer las suposiciones que estime necesarias, como el número de pistas de la lista de reproducción. Cuando termine, compare su código con el que encontrará en el apartado 5.3.

---

# 3. Interbloqueos

El interbloqueo es una situación no deseada en la que dos o más hilos se encuentran bloqueados entre sí porque esperan que ocurra algo, que no va a suceder, para ser liberados. Normalmente, un hilo espera que otro cumpla una condición que liberaría al primero, y viceversa.

Hay que tener mucho cuidado para evitar las referencias cruzadas y los interbloqueos. Para ello, se debe analizar cuidadosa y rigurosamente el código sincronizado.

A continuación podemos ver un ejemplo de interbloqueo entre dos hilos. Suponga una mesa de trabajo donde trabajan 2 personas. Cada una de ellas debe hacer un determinado número de líneas en un papel y, además, comparten los materiales que necesitan. Representaremos los materiales con una sencilla clase:

```
/**
 * Clase Objeto.
 *
 * @author Natalia Partera
 * @version 1.0
 */

public class Objeto {

  private String nombre;

  //Constructor de la clase
  public Objeto(String obj) {
    nombre = obj;
  }

  //Método observador que devuelve el nombre del objeto
  public String Nombre() {
    return nombre;
  }
}
```

Los trabajadores pueden coger los materiales en cualquier orden. La situación problemática sería que cada uno coja un material distinto al principio y que al intentar coger el otro, ya lo haya cogido otra persona. Para asegurarnos de que esta situación se de, podemos definir dos clases distintas, o, como los materiales pertenecen a la misma clase, hacer la siguiente clase genérica y cambiar el orden en que los reciben en la llamada a su constructor:

```
/**
 * Clase Trabajador.
 *
 * @author Natalia Partera
 * @version 1.0
 */

public class Trabajador extends Thread {
  private String nombre;
  private Objeto objeto1;
  private Objeto objeto2;

  //Constructor
  public Trabajador(String n, Objeto obj1, Objeto obj2) {
    nombre = n;
    objeto1 = obj1;
    objeto2 = obj2;
```

```
  }

  //Código que ejecuta cada hilo
  public void run() {
    synchronized(objeto1) {
      for(int i = 0; i < 1000; ++i) {
        synchronized(objeto2) {
          System.out.println(nombre + ": ==========");
        }
      }
    }
  }
}
```

Otra opción para que los trabajadores cojan los materiales aleatoriamente, sería ponerlos en una estructura (un vector, una lista, un conjunto, una cola, ...) y que cojan el primero que encuentren libre. Para probar las clases anteriores, usamos el siguiente programa:

```
/**
 * Programa que representa el funcionamiento de una oficina con materiales
 * compartidos.
 *
 * @author Natalia Partera
 * @version 1.0
 */

public class Oficina {
  //Programa principal
  public static void main (String[] args) {
    final Objeto reglas = new Objeto("Reglas");
    final Objeto boli = new Objeto("Boligrafo");

    Trabajador t1 = new Trabajador("Pepe", reglas, boli);
    Trabajador t2 = new Trabajador("Juan", boli, reglas);

    t1.start();
    t2.start();
  }
}
```

Puede que en una primera ejecución este código no se bloquee. Pero al ejecutar varias veces o aumentar las vueltas del método `run()`, seguramente llegue a interbloqueo.

---

**Ejercicio 5** Ejecute el ejemplo anterior hasta que se produzca interbloqueo. A continuación, solucione el problema del interbloqueo. Compare su solución con la que se indica en el apartado 5.4.

---

# 4. Sincronización

A veces el método proporcionado por `synchronized` no es el más eficiente para conseguir la sincronización entre varios hilos. Un caso muy claro es cuando en el método `run()` de un hilo, existe un bucle que comprueba constantemente el estado del cerrojo. El programa mejoraría si en vez de estar comprobando esta condición continuamente, existiera una comunicación en el programa para poder avisar al hilo cuando la condición que espera se haya cumplido.

Este mecanismo existe, se trata de las notificaciones de hilos. Consiste principalmente en dos tipos de métodos: uno que bloquea el hilo hasta que no sea avisado y otro que avisa a los hilos en espera de que una condición se ha cumplido. Los métodos que nos proporcionan este funcionamiento son `wait()` y `notify()`. Estos métodos pertenecen a la clase `Object`, por lo que pueden ser usados por cualquier clase (recordemos que toda clase en Java hereda de `Object`).

El funcionamiento de este mecanismo es el siguiente: cuando un hilo tiene que esperar a que se cumpla una condición, se utiliza la función `wait()` para ponerlo en espera. El hilo se añade a una cola "`wait-set`" de hilos en espera asociados al cerrojo de un objeto. Cuando en otra parte del programa se cumple la condición de espera de un hilo, se notifica con `notify()`, que despierta a un hilo de la cola de hilos en espera. Pero este mecanismo no sabe comunicar cuál es la condición que se ha cumplido. Cuando para ese objeto sólo hay un hilo en espera, puede ser suficiente con usar `notify()`, pero si ese no es nuestro caso, podemos usar `notifyAll()`, que despierta a todos los hilos de la cola para avisarles que una condición de espera se ha cumplido. El inconveniente de este método es que se pueden despertar hilos que no deban. Así pues, lo recomendable es cada hilo compruebe si su condición se ha cumplido cuando se despierte. En el caso de que su condición no haya sido la notificada, hay que poner a dicho hilo en espera otra vez. El siguiente diagrama representa el funcionamiento del mecanismo:

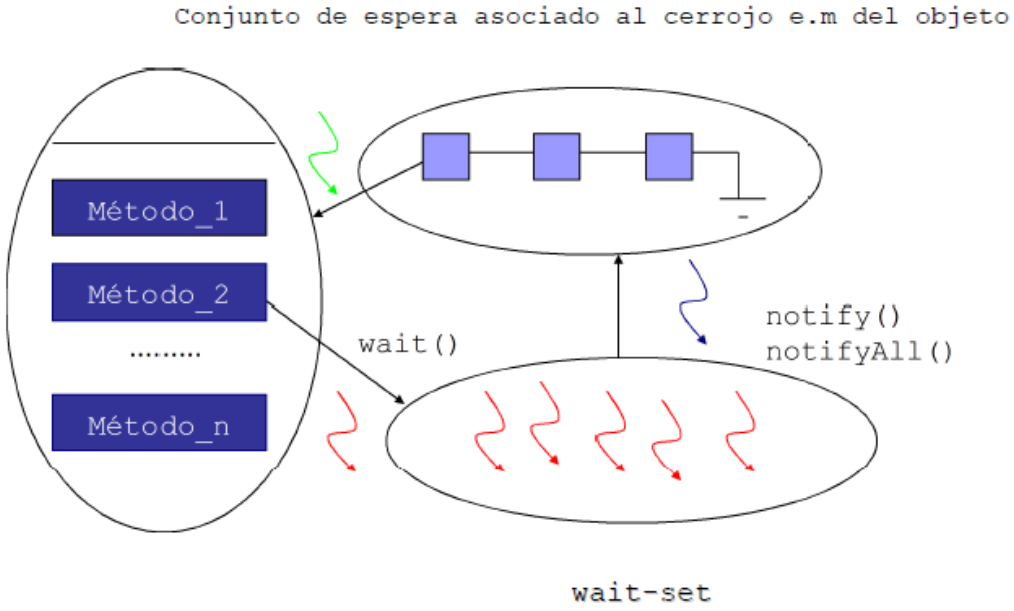


Figura 3: Imagen que muesta el comportamiento del mecanismo `wait-notify`.

Para una correcta sincronización, los métodos `wait()` y `notify()` sólo deben ser utilizados dentro de métodos o bloques `synchronized`. El mecanismo `wait-notify` no resuelve el problema de que el resultado del programa cambie según el orden de la ejecución de los hilos, problema que el mecanismo de sincronización sí resuelve.

Aunque antes la utilización de la etiqueta `synchronized` no era recomendada para algunos métodos, ahora la combinación de la etiqueta `synchronized` y del mecanismo `wait-set` resuelve los problemas causados antes. Internamente, el método `wait()` realiza las siguientes acciones atómicamente:

1. El hilo que llama a `wait()` pasa a estado suspendido y bloqueado.
2. El cerrojo del objeto es liberado.
3. El hilo es colocado en una cola única (`wait-set`) de espera asociada al objeto.

Cuando un hilo invoca a `notify()`, uno de los hilos de la cola de espera del objeto pasa a listo. Pero Java no especifica cuál es, dependerá de la implementación de la máquina virtual (JVM). Por tanto, muchas veces lo recomendable es despertarlos a todos y que cada uno compruebe su condición. Si se hiciera una llamada a `notify()` o a `notifyAll()` y no hubiera ningún hilo en espera, no ocurre nada.

Veamos a continuación un ejemplo de sincronización usando `wait` y `notify`. Tenemos una clase `Contador` que representa un número que será incrementado por varios hilos.

```
/**
 * Clase Contador.
 *
 * @author Natalia Partera
 * @version 1.0
 */

public class Contador {

  //Atributos privados
  private int num = 0;

  //Constructor
  public Contador() {}

  //Métodos
  public synchronized void aumentar() {
    ++num;
  }

  public synchronized void aumentar(int cant) {
    num = num + cant;
  }

  public synchronized int ver() {
    return num;
  }

}
```

Cada hilo pertenecerá a la clase `Cuenta`. Podemos elegir que cada hilo sume de 1 en 1 o de 5 en 5, por ejemplo.

```
/**
 * Clase Cuenta.
 *
 * @author Natalia Partera
 * @version 1.0
 */

public class Cuenta extends Thread {

  //Atributos privados
  private static Contador cont;
  private int num_min;
  private int num_max;
  private int constante;

  //Constructor
  public Cuenta() {}
  public Cuenta(Contador c, int min, int max, int cte) {
    cont = c;
    num_min = min;
    num_max = max;
    constante = cte;
  }

  //Métodos
  public void run() {
    synchronized(cont) {
      while(cont.ver() < num_max) {
        while(cont.ver() < num_min || cont.ver() > num_max) {
          try {
            wait();
          } catch (Exception e) {}
        }
        cont.aumentar(constante);
        System.out.print(cont.ver() + " ");
        cambiarHilo();
      }
    }
  }

  public synchronized void cambiarHilo() {
    if(cont.ver() >= num_max) {
      notifyAll();
      this.stop();
    }
  }

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

```
    Contador cont = new Contador();

    Cuenta hilo1 = new Cuenta(cont, 0, 50, 1);
    Cuenta hilo2 = new Cuenta(cont, 50, 150, 5);

    hilo1.start();
    hilo2.start();

    hilo1.join();
    hilo2.join();
    System.out.println();
  }

}
```

Al ejecutar el ejemplo, puede ver como el primer hilo aumenta el contador de uno en uno. Cuando el primer hilo llega a su tope, continúa el otro hilo aumentando el contador de 5 en 5.

---

**Ejercicio 6** Modifique el ejemplo de la centralita de teléfonos del apartado 2.2. En esta ocasión, sólo podrá haber un teléfono recibiendo o enviando una llamada a la vez. Controle esta condición usando las notificaciones de hilo para sincronizar el comportamiento. Aumente el número de llamadas de cada teléfono para observar mejor la sincronización entre éstos. Cuando acabe, puede comprobar su código con el que encontrará en el apartado 5.5.

---

# 5. Soluciones de los ejercicios

En esta sección encontrará las soluciones a los ejercicios propuestos a lo largo del presente guión.

## 5.1. Ejercicio 2

Para aumentar la cuenta atrás tan sólo hay que cambiar el valor del constructor del objeto `CuentaAtras` en la clase `UsaCuentaAtras`. Tras ello la clase debe quedar así:

```
/**
 * Programa que realiza una cuenta atrás usando varios hilos.
 *
 * @author Natalia Partera
 * @version 1.0
 */

public class UsaCuentaAtras extends Thread {

  //Variable compartida que debe ser ejecutada en EM.
  private static CuentaAtras cont;

  //Código que ejecuta cada hilo
  public void run() {
    while(cont.valor() > 0) {
      cont.contar();
    }
  }

  //Programa principal
  public static void main (String[] args) {
    cont = new CuentaAtras(50);
    UsaCuentaAtras hilo1 = new UsaCuentaAtras();
    UsaCuentaAtras hilo2 = new UsaCuentaAtras();
    UsaCuentaAtras hilo3 = new UsaCuentaAtras();

    hilo1.start();
    hilo2.start();
    hilo3.start();
  }

}
```

En el ejemplo con 35 números puede que no se notase la interfoliación de instrucciones, pero con 50 debería notarse. Si no fuera así, siga aumentando el número. Debe detectar en la salida algo como lo siguiente:

```
[...]
Actualizando...
Contador actualizado: 43
Actualizando...
Variable actualizada.
[...]
```

Para evitar estos casos, hay que introducir el mensaje `Variable actualizada.` en el bloque `synchronized`. La clase `CuentaAtras` quedaría así:

```
/**
 * Clase CuentaAtras.
 *
 * @author Natalia Partera
 * @version 1.0
 */

public class CuentaAtras {

  //Atributo privado
  private int cont;

  //Constructor de la clase
  public CuentaAtras(int num) {
    if(num > 0)
      cont = num;
    else
      cont = 0;
  }

  //Método modificador que decrementa la variable
  public void contar() {
    System.out.println("Actualizando...");
    synchronized(this) {
      if(cont > 0) {
        System.out.println("Contador actualizado: " + cont);
        --cont;
      }
      System.out.println("Variable actualizada.");
    }
  }

  //Método observador que devuelve el valor de la variable
  public int valor() {
    synchronized(this) {
      return cont;
    }
  }

}
```

## 5.2. Ejercicio 3

Esta es la solución más básica para el ejercicio 3. En ella, no se permite cancelar citas ni se listan. A continuación puede ver la clase que representa las citas disponibles:

```
/**
 * Clase Citas.
```

```
 *
 * @author Natalia Partera
 * @version 1.0
 */

public class Citas {

  //Atributos privados
  private int asignadas;
  private int numCitas = 10;

  //Constructor de la clase
  public Citas() {}
  public Citas(int num) {
    numCitas = num;
  }

  //Método modificador que asigna una cita al cliente que lo invoca
  public int asignarCita() {
    synchronized(this) {
      int num;
      if(numCitas > asignadas) {
        ++asignadas;
        num = asignadas;
      }
      else {
        System.out.println("Lo sentimos, no quedan citas.");
        num = -1;
      }
      return num;
    }
  }

  //Método observador que indica si hay citas disponibles
  public int citasDisponibles() {
    return (numCitas - asignadas);
  }

  //Método observador que muestra las citas disponibles
  public void verCitasDisponibles() {
    System.out.println("Quedan " + (numCitas - asignadas) + " citas libres para
      hoy.");
  }
}
```

El comportamiento de los clientes y el programa de prueba se situan en la clase `Cliente`:

```
/**
 * Clase Cliente.
 *
 * @author Natalia Partera
 * @version 1.0
```

```
 */

public class Cliente extends Thread {

  //Atributo privado
  private int cita;
  private static Citas citas;

  //Constructor de la clase
  public Cliente() {}
  public Cliente(Citas cit) {
    citas = cit;
  }

  //Método modificador
  public void pedirCita() {
    synchronized(citas) {
      if(citas.citasDisponibles() > 0) {
        cita = citas.asignarCita();
        System.out.println("La cita asignada es la número " + cita + ".");
      }
      else
        System.out.println("Mejor, para otro día.");
        this.stop();
    }
  }

  //Código que ejecuta cada cliente
  public void run() {
    while(cita == 0) {
      this.pedirCita();
    }
  }

  //Programa principal
  public static void main (String[] args) {
    citas = new Citas(5);

    Cliente [] clientes = new Cliente[6];
    for(int i = 0; i < 6; ++i) {
      clientes[i] = new Cliente(citas);
      clientes[i].start();
    }
  }

}
```

## 5.3. Ejercicio 4

Para simular el comportamiento de un reproductor de música, crearemos dos clases, uno que emule el propio reproductor y el funcionamiento de sus botones, y otra que simule una ejecución de ejemplo. A

continuación puede ver la clase que representa al reproductor:

```
/**
 * Clase Reproductor.
 *
 * @author Natalia Partera
 * @version 1.0
 */

public class Reproductor {

  //Atributos privados
  private int id_track;
  private boolean pause = false;
  private boolean playing = false;
  private static final int MAX_TRACK = 22;

  //Constructor
  public Reproductor() {}
  public Reproductor(int pista) {
    id_track = pista;
  }

  //Métodos
  public synchronized void Reproducir() {
    if (!playing) {
      if (id_track == 0) {
        id_track = 1;
        System.out.println("Reproduciendo pista " + id_track + ".");
      }
      else if (id_track < 0 || id_track > MAX_TRACK) {
        id_track = 1;
        System.out.println("Pista no encontrada. Reproduciendo desde el principio:
          pista " + id_track);
      }
      pause = false;
      playing = true;
    }
  }

  public synchronized void Reproducir(int pista) {
    if (pista > 0 && pista <= MAX_TRACK) {
      id_track = pista;
      System.out.println("Reproduciendo pista seleccionada. Reproduciendo pista " +
        id_track + ".");
      pause = false;
      playing = true;
    }
  }

  public synchronized void Pausar() {
    if(playing == true) {
```

```
      if (pause == false) {
        pause = true;
        System.out.println("Reproducción en pausa: pista " + id_track);
      }
      else {
        pause = false;
        System.out.println("Reanudando reproducción: pista " + id_track);
      }
    }
  }

  public synchronized void Parar() {
    if(playing == true) {
      pause = false;
      playing = false;
      System.out.println("Reproducción parada.");
    }
    id_track = 0;
  }

  public synchronized void Anterior() {
    if(id_track == 0) {
      id_track = MAX_TRACK;
    }
    --id_track;
    System.out.println("Cambio a la pista anterior. Pista actual: " + id_track);
  }

  public synchronized void Siguiente() {
    if(id_track == MAX_TRACK) {
      id_track = 0;
    }
    ++id_track;
    System.out.println("Cambio a la siguiente pista. Pista actual: " + id_track);
  }

}
```

Para probar el funcionamiento de los métodos sincronizados, crearemos un programa de prueba que invocará el funcionamiento de los botones y los programará para que se repitan. Tras simular la pulsación de algunos botones, el programa terminará.

```
/**
 * Programa que simula el manejo de un reproductor de música.
 *
 * @author Natalia Partera
 * @version 1.0
 */

public class UsaReproductor implements Runnable {
```

```
//Objeto compartido
static Reproductor rep;
int button;
static int num_oper = 0;

//Constructor
UsaReproductor(Reproductor ref, int boton) {
  rep = ref;
  button = boton;
}

//Método run
public void run() {
  try {
    while(num_oper < 10) {
      switch(button) {
        case 1:
          if(num_oper%2 == 0)
            rep.Reproducir();
          else
            rep.Reproducir(num_oper);
          break;
        case 2:
          Thread.sleep(4500);
          rep.Pausar();
          break;
        case 3:
          Thread.sleep(10000);
          rep.Parar();
          break;
        case 4:
          Thread.sleep(6000);
          rep.Anterior();
          break;
        case 5:
          Thread.sleep(2000);
          rep.Siguiente();
          break;
      }
      ++num_oper;
      Thread.sleep(2500);
    }
  } catch(InterruptedException ie) {
      ie.printStackTrace();
      System.exit(-1);
  }
}

//Programa principal
public static void main (String[] args) {
  Reproductor reproductor = new Reproductor();
```

```
    //Botón reproducir
    Thread bot1 = new Thread(new UsaReproductor(reproductor, 1));
    //Botón pausar
    Thread bot2 = new Thread(new UsaReproductor(reproductor, 2));
    //Botón parar
    Thread bot3 = new Thread(new UsaReproductor(reproductor, 3));
    //Botón anterior
    Thread bot4 = new Thread(new UsaReproductor(reproductor, 4));
    //Botón siguiente
    Thread bot5 = new Thread(new UsaReproductor(reproductor, 5));

    bot1.start();
    bot2.start();
    bot3.start();
    bot4.start();
    bot5.start();
  }
}
```

## 5.4. Ejercicio 5

Para solucionar el problema del interbloqueo, basta con cuidar el orden de las acciones. Si ambos trabajadores cogen siempre el mismo objeto primero, evitamos fácilmente el interbloqueo. Para conseguir esto, basta cambiar en nuestro programa principal el orden en que se le pasan los materiales al constructor de los trabajadores:

```
/**
 * Programa que representa el funcionamiento de una oficina con materiales
 * compartidos.
 *
 * @author Natalia Partera
 * @version 1.0
 */

public class Oficina {

  //Programa principal
  public static void main (String[] args) {
    final Objeto reglas = new Objeto("Reglas");
    final Objeto boli = new Objeto("Boligrafo");

    Trabajador t1 = new Trabajador("Pepe", reglas, boli);
    Trabajador t2 = new Trabajador("Juan", reglas, boli);

    t1.start();
    t2.start();
  }

}
```

## 5.5. Ejercicio 6

Para sincronizar las llamadas y que sólo se pueda recibir o enviar una a la vez en cualquier teléfono, creamos una nueva clase `Linea`. Esta clase `Linea` representa la línea telefónica, que será el recurso compartido por los teléfonos.

```
/**
 * Clase Linea.
 *
 * @author Natalia Partera
 * @version 1.0
 */

public class Linea {
  private boolean linea_ocupada;

  public Linea() {
    linea_ocupada = false;
  }

  public synchronized void CambiarEstado() {
    if(linea_ocupada)
      linea_ocupada = false;
    else
      linea_ocupada = true;
  }

  public synchronized boolean LineaOcupada() {
    return linea_ocupada;
  }
}
```

En la clase `Telefono` añadimos sincronización usando las notificaciones de hilos. Además, aumentamos el número de llamadas.

```
/**
 * Clase Telefono.
 *
 * @author Natalia Partera
 * @version 1.0
 */

public class Telefono implements Runnable{

  //Atributos privados
  private int id;
  private Linea linea;

  //Constructor
  public Telefono() {}
  public Telefono(int id, Linea lin) {
```

```
    this.id = id;
    linea = lin;
  }

  //Métodos
  public synchronized void LlamadaEntrante() {
    System.out.println("Teléfono " + id + ": Iniciando llamada entrante.");
    for(int i = 0; i < 500; ++i) {}
    System.out.println("Teléfono " + id + ": Llamada entrante en curso.");
    for(int i = 501; i < 1000; ++i) {}
    System.out.println("Teléfono " + id + ": Finalizando llamada entrante.");
    linea.CambiarEstado();
    notify();
  }

  public synchronized void LlamadaSaliente() {
    System.out.println("Teléfono " + id + ": Iniciando llamada saliente.");
    for(int i = 0; i < 500; ++i) {}
    System.out.println("Teléfono " + id + ": Llamada saliente en curso.");
    for(int i = 501; i < 1000; ++i) {}
    System.out.println("Teléfono " + id + ": Finalizando llamada saliente.");
    linea.CambiarEstado();
    notify();
  }

  //Método run()
  public void run() {
    for(int i = 0; i < 15; ++i) {
      synchronized(linea) {
        while(linea.LineaOcupada()) {
          try {
            wait();
          } catch(Exception e) {}
        }
        linea.CambiarEstado();
        if(i%2 == 0) {
          LlamadaEntrante();
        }
        else {
          LlamadaSaliente();
        }
      }
    }
  }

}
```

Por último, con el siguiente código podemos probar el programa.

```
/**
 * Programa que simula el comportamiento de una centralita con varios teléfonos.
```

```
 *
 * @author Natalia Partera
 * @version 1.0
 */

public class Centralita {

  //Programa principal
  public static void main (String[] args) {
    Linea linea = new Linea();

    Thread tel1 = new Thread(new Telefono(1, linea));
    Thread tel2 = new Thread(new Telefono(2, linea));
    Thread tel3 = new Thread(new Telefono(3, linea));

    tel1.start();
    tel2.start();
    tel3.start();
  }

}
```