Unidad 2. Programación multihilo
Parte 2. Sincronización y coordinación de hilos en Java

Existen diferentes formas de sincronizar o coordinar hilos cuando se lanzan desde una misma aplicación. Podemos, por ejemplo, asignar distintas prioridades a cada hilo para que algunos sean más rápidos que el resto. También podemos "unir" hilos, es decir, hacer que un hilo espere hasta que otro hilo termine su tarea por completo. A partir de ahí, existen estructuras de sincronización más complejas como la exclusión mutua, los bloqueos... Algunas de estas técnicas las veremos en esta parte.

1. Coordinación básica. Unión de hilos

Si queremos que un hilo espere hasta que otro hilo termine, podemos utilizar el método join del hilo al que queremos esperar. En este ejemplo, la aplicación principal crea un hilo y espera hasta que éste termine antes de continuar:

```
public static void main(String[] args) {
    Thread t = new MyThread();
    t.start();
    t.join();
}
```

De hecho, el método `join` puede generar una `InterruptedException`, por lo que debemos capturarla:

```
public static void main(String[] args) {
    Thread t = new MyThread();
    t.start();
    try {
        t.join();
    } catch (InterruptedException e) {
        ...
    }
}
```

Si queremos que un hilo secundario (no el programa principal) espere a otro hilo, entonces debemos indicarle a este hilo cuál es el hilo al que debe esperar. Normalmente, usamos un atributo dentro de la clase del hilo para almacenar esta información:

```java
public class MyThread extends Thread {
    Thread waitThread;

    // We will use this constructor
    // if thread does not have to wait for anyone
    public MyThread() {
        waitThread = null;
    }

    // We will use this constructor
    // if thread has to wait for thread "wt"
    public MyThread(Thread wt) {
        waitThread = wt;
    }

    // We check if waitThread attribute is not null,
    // and then call the join method before keep on running
    public void run() {
        if (waitThread != null)
            waitThread.join();
        ...
    }
}
```

Luego, en la aplicación principal creamos dos hilos de tipo MyThread y le pedimos a uno de ellos que espere al otro:

```java
public static void main(String[] args) {
    Thread t1 = new MyThread();
    Thread t2 = new MyThread(t1);
    t1.start();
    // We start thread t2, but it will not run until t1 finishes
    t2.start();
}
```

Ejercicio 1:
Crea un proyecto llamado **ThreadRaceJoin** basado en el proyecto anterior del *Ejercicio 3*. Cambia el comportamiento de los tres hilos en ejecución (A, B y C) para que cada uno comience a ejecutarse cuando el hilo anterior haya finalizado:

- El hilo A comenzará al principio del programa.
- El hilo B comenzará cuando el hilo A termine.
- El hilo C comenzará cuando el hilo B termine.
- El programa principal esperará hasta que el último hilo (C) termine la carrera.

Ejercicio 2:
Crea un proyecto llamado **MultiplierThreadsJoin** basado en el proyecto anterior del *Ejercicio 2*. Cambia el comportamiento de la aplicación principal para que espere a que cada hilo termine antes de comenzar el siguiente. Por lo tanto, todas las tablas de multiplicar se mostrarán en orden:

```
0 x 0 = 0
0 x 1 = 0
...
0 x 10 = 0
1 x 0 = 0
...
```

2. Acceso a recursos compartidos. La necesidad de sincronizar hilos

Es bastante habitual que varios hilos quieran obtener un mismo recurso (p. ej. acceso a una variable, un archivo de texto, una base de datos...), y es difícil garantizar que la información de ese recurso no se modifique por error (por ejemplo, que un hilo cambie el valor de una variable mientras otro hilo la está utilizando). El fragmento de código que se encarga de permitir que los hilos obtengan ese recurso compartido se denomina comúnmente **sección crítica**. Este código no debe ser ejecutado por más de un hilo a la vez. Para lograrlo, Java ofrece algunas opciones.

Veamos el problema en profundidad con este ejemplo: en primer lugar, creamos un objeto de la clase `Counter`, que será compartido entre hilos:

```java
public class Counter {
    int value;

    public Counter(int value) {
        this.value = value;
    }

    public void increment() {
        value++;
    }

    public void decrement() {
        value--;
    }

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

Puedes ver que la clase `Counter` tiene solo un atributo, *value*, que es el valor que será leído y/o modificado por los hilos, al llamar a los métodos `increment` o `decrement`.

Luego creamos dos tipos de hilos: uno que incrementará el valor del objeto en un bucle y otro que lo disminuirá:

```
public static void main(String[] args) {
    Counter c = new Counter(100);

    Thread tinc = new Thread(() -> {
        for (int i = 0; i < 100; i++) {
            c.increment();
            try {
                Thread.sleep(100);
            } catch (InterruptedException e) { }
        }
        System.out.println("Finishing inc. Final value = "+c.getValue());
    });

    Thread tdec = new Thread(() -> {
        for (int i = 0; i < 100; i++) {
            c.decrement();
            try {
                Thread.sleep(100);
            } catch (InterruptedException e) { }
        }
        System.out.println("Finishing dec. Final value = "+c.getValue());
    });
    tinc.start();
    tdec.start();
}
```

¿Qué sucederá? Si pruebas este ejemplo en tu IDE, descubrirás que el valor final del objeto `c` es diferente cada vez que ejecuta el ejemplo. A veces será 105, a veces será 97... pero siempre debería ser 100 (comienza con 100 y luego se espera que un hilo incremente el valor 100 veces y que el otro hilo también lo disminuya 100 veces).

¿Por qué puede suceder esto? Bueno, puede ocurrir que `tinc` entre en el método `increment` y luego el control vaya a `tdec`, que entra en el método `decrement`. Entonces, una de estas operaciones (ya sea `value++` o `value--`) no tendrá ningún efecto. Por ejemplo, si `tinc` lee el valor 100 e intenta establecerlo en 101 pero luego el control va a `tdec` que lee el mismo valor 100 (`tinc` aún no lo ha cambiado) y lo establece en 99, entonces cuando el control regrese a `tinc`, establecerá value en 101, y el decremento habrá desaparecido, perdiendo información.

Para solucionar este problema, Java ofrece algunos mecanismos que pueden ser utilizados por un hilo para comprobar si hay algún otro hilo ejecutando la sección crítica antes de entrar en ella. Si es así, el hilo que intenta entrar en la sección crítica es suspendido por el mecanismo de sincronización. Si hay más de un hilo esperando a que un hilo termine la sección crítica, tan pronto como esta termina, la JVM elige uno de los hilos en espera (al azar) para ejecutarlo. Veamos cómo funciona este mecanismo y sus variaciones.

2.1. Métodos de sincronización

Uno de los métodos más básicos de sincronización en Java es usando un monitor mediante la palabra clave `synchronized`. Podemos utilizarla para controlar el acceso a un método, de forma

que se convierta en una sección crítica. Java sólo permite la ejecución de una sección crítica en cada objeto. Si el método es estático, entonces esta sección crítica es independiente de todos los objetos de esa clase. En otras palabras, Java sólo permite la ejecución de una sección crítica por objeto, y una sección crítica estática por clase.

En el ejemplo anterior, si simplemente agregamos la palabra clave `synchronized` a los métodos `increment` y `decrement` de la clase Counter:

```
public class Counter {
    int value;

    public Counter(int value) {
        this.value = value;
    }

    public synchronized void increment() {
        value++;
    }

    public synchronized void decrement() {
        value--;
    }

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

y volvemos a ejecutar el programa, notaremos que ahora funciona perfectamente. ¿Por qué? Bueno, si `tinc` entra en el método `increment`, entonces `tdec` no podrá entrar en el método `decrement`, y viceversa, por lo que no tendremos el problema de incrementar y decrementar el valor al mismo tiempo, porque ambos hilos comparten el mismo objeto `Counter`, y solo un hilo puede estar ejecutando un método *synchronized* al mismo tiempo.

También puedes notar que el programa se ejecuta más lento que antes; lo cual es uno de los efectos de la sincronización, ya que penaliza el rendimiento de la aplicación porque detiene la ejecución de los hilos en algunos momentos.

Ejercicio 3:

Crea un proyecto llamado **BankAccountSynchronized** con estas clases y métodos:

- Una clase `BankAccount` con un atributo llamado `balance` que almacenará cuánto dinero hay en la cuenta. Agrega un constructor para inicializar el dinero en la cuenta y dos métodos `addMoney` y `takeOutMoney`, que sumarán o quitarán la cantidad pasada como parámetro. Añade también un método `getBalance` para recuperar el saldo actual de la cuenta.

```
public BankAccount(int balance) { ... }
public void addMoney(int money) { ... }
public void takeOutMoney(int money) { ... }
public int getBalance() { ... }
```

- Una clase `BankThreadSave` con un atributo de tipo `BankAccount`. Puedes extender la clase `Thread` o implementar la interfaz `Runnable` para hacer esa clase. En el método "`run`" , el hilo agregará 100 € a la cuenta bancaria 5 veces, con un intervalo de 100 ms entre cada operación.
- Una clase `BankThreadSpend` con un atributo de tipo `BankAccount`. Puedes extender la clase `Thread` o implementar la interfaz `Runnable` para hacer esa clase. En el método "`run`", el hilo extraerá 100 € de la cuenta bancaria 5 veces, durmiendo 100 ms entre cada operación.
- Desde la clase principal, crea un objeto `BankAccount` y una matriz de 20 objetos `BankThreadSave` y 20 objetos `BankThreadSpend`, utilizando todos ellos el mismo objeto `BankAccount`. Inícialos todos y observa cómo cambia el saldo de la cuenta bancaria (muestra un mensaje en algún lugar para mostrar el saldo después de cada operación).
- En este punto, ya te habrás dado cuenta de que tu cuenta bancaria no funciona correctamente. Añade los mecanismos de sincronización que consideres para solucionar el problema.

2.2 Sincronización de objetos

También podemos aplicar la palabra clave `synchronized` a un objeto dado en un fragmento de código, pasando el objeto como parámetro de esta manera:

```
public void myMethod() {
    int someValue;
    ...
    synchronized(this) {
        someValue++;
        System.out.println("Value changed: " + someValue);
    }
    ...
}
```

Entonces, cuando un hilo A intenta ejecutar las instrucciones dentro de este bloque, no podrá hacerlo si otro hilo B ya está ejecutando una sección crítica que afecta al objeto `this`. Tan pronto como este hilo B termine la sección crítica, el otro hilo A se despertará y entrará en la sección crítica. Por supuesto, podemos usar cualquier otro objeto con la palabra clave `synchronized`.

Por ejemplo, si tenemos un objeto llamado `file` y queremos crear una sección crítica a su alrededor, podemos hacerlo de esta manera:

```
public void someMethod() {
    ...
    synchronized(file)     {
        ... // Critical section
    }
    ...
}
```

Ejercicio 4:
Crea un proyecto **BankAccountSynchronizedObject** basado en el ejercicio anterior. En este caso no se puede sincronizar ningún método, solo puedes sincronizar objetos. ¿Qué cambios agregarías al proyecto para asegurarte de que siga funcionando correctamente?

3. Prioridades de los hilos

Cuando tenemos varios hilos ejecutándose en un programa podemos cambiar la prioridad de cada hilo, de modo que aquellos hilos con mayor prioridad obtengan el procesador con mayor frecuencia. Esta característica solo se aplica a los hilos, no a los procesos, ya que la JVM no puede intervenir en los procesos externos que no dependen de ella (otros procesos en ejecución en el SO).

Las prioridades en los hilos son simplemente números enteros desde 1 (almacenado en la constante `Thread.MIN_PRIORITY`) hasta 10 (almacenado en la constante `Thread.MAX_PRIORITY`). De manera predeterminada, cada hilo tiene una prioridad de 5 (`Thread.NORM_PRIORITY`) y cada hilo hereda la prioridad de su padre, a menos que la cambiemos más adelante.

Para cambiar la prioridad de un hilo podemos utilizar el método `setPriority` pasando un entero como parámetro. También podemos consultar la prioridad del hilo con `getPriority`.

```
Thread t1 = new MyThread();
Thread t2 = new MyThread();
Thread t3 = new MyThread();

t1.setPriority(Thread.MIN_PRIORITY);
t2.setPriority(Thread.NORM_PRIORITY);
t3.setPriority(Thread.MAX_PRIORITY);

System.out.println("Priority of thread #2 is " + t2.getPriority());
```

3.1 Dependencia del sistema operativo

Existe un problema con las prioridades según el sistema operativo que estemos usando. En sistemas Windows, verás que tus hilos se comportan más o menos según sus prioridades, pero en sistemas Linux y Mac OS X, la prioridad que intentemos establecer para cada hilo no tiene ningún efecto. Por lo que debemos tener en cuenta que el comportamiento esperado de nuestros hilos no está garantizado, y dependerá del sistema operativo, a menos que busquemos otras opciones.

Si necesitamos asegurarnos de que algunos hilos tendrán una prioridad más alta, no podemos confiar en el método `setPriority`, ya que el sistema operativo puede ignorar estas prioridades. Una opción alternativa es utilizar números aleatorios y los métodos `yield` ó `sleep` para obligar a los hilos a liberar el procesador según su prioridad real. Veamos este ejemplo:

```
public class MyPrioritizedThread extends Thread {
    int priority;

    public MyPrioritizedThread(int priority) {
        this.priority = priority;
    }

    @Override
    public void run() {
        java.util.Random r =
            new java.util.Random(System.currentTimeMillis());

        while (condition) {
            // Generate a random number between 1 and 10
            int number = r.nextInt(10) + 1;

            // If this number is greater or equal than thread's
            // priority, yield
            if (number >= priority)
                Thread.yield();

            ... // Rest of the instructions for our run loop
        }
    }
}
```

Definimos nuestra subclase `Thread` con su propio atributo `priority`, que será administrado por nuestro código. En el método `run`, generamos un número aleatorio entre 1 (`Thread.MIN_PRIORITY`) y 10 (`Thread.MAX_PRIORITY`). Si este número es mayor o igual que el atributo `priority`, entonces nuestro hilo cederá. Observa que los hilos con prioridades más bajas (es decir, más cercanas a 1) cederán con mayor frecuencia, y los hilos con prioridades más altas (es decir, más cercanas a 10) cederán "de vez en cuando".

Si el planificador de tareas ignora la instrucción `yield` y nuestras prioridades no parecen tener ningún efecto, entonces reemplaza la instrucción `yield` con un tiempo de sueño:

```
if (number >= priority)
    try {
        Thread.sleep(5);
    } catch (Exception e) {}
```

Cuantos más milisegundos pongas a dormir a tus hilos, más tiempo tardarán los hilos con prioridades más bajas en finalizar su tarea. Depende de ti ajustar la cantidad de milisegundos más adecuada, según la aplicación que estés desarrollando.

Ejercicio 5:
Crea un proyecto llamado **ThreadRacePriorities** basado en el proyecto *ThreadRace* anterior. Modifica el código para que el hilo *A* tenga `MAX_PRIORITY`, el hilo *B* tenga `NORM_PRIORITY` y el hilo *C* tenga `MIN_PRIORITY`. Hazlo con el método `setPriority()`. Intenta comprobar los resultados en diferentes sistemas operativos.

4. El problema productor-consumidor

El problema productor-consumidor es un problema clásico en programación concurrente. En este tipo de problemas tenemos un buffer de datos, algunos productores que introducen datos en ese buffer y algunos consumidores que obtienen datos del buffer. Tenemos que asegurarnos de que los consumidores no intenten obtener datos cuando el buffer esté vacío y, en algunos casos, que los productores no produzcan más datos hasta que los consumidores procesen el existente, o si el buffer está lleno. En este tipo de problemas el uso de la palabra clave `synchronized` no es suficiente, tenemos que añadir algunos mecanismos para hacer que los productores o consumidores esperen hasta que la otra parte haya hecho su trabajo. Para ello, podemos utilizar los métodos `wait()`, `notify()` y `notifyAll()` de la clase `Object`:

- El método `wait()` se puede llamar dentro de un bloque sincronizado. Entonces, la JVM pone el hilo en reposo y libera el objeto bloqueado por este bloque sincronizado, de modo que otros hilos que ejecutan bloques sincronizados sobre el mismo objeto puedan continuar.
- Los métodos `notify()` o `notifyAll()` son llamados por un hilo que ha terminado su tarea dentro de una sección crítica antes de salir de ella, para indicar a la JVM que puede despertar a un hilo que previamente había sido puesto a dormir con un método `wait()`. La principal diferencia entre estos dos métodos (`notify` y `notifyAll`) es que, con `notify`, la JVM elige un hilo que está esperando (de forma aleatoria), mientras que con `notifyAll` la JVM despierta todos los hilos que están esperando, y el primero que entra en la sección crítica es el que sigue (los demás seguirán esperando).

Veamos un ejemplo: crearemos dos tipos de hilos: un `Producer` que colocará algunos datos (por ejemplo, un entero) en un objeto compartido dado (lo llamaremos `SharedData`), y un `Consumer` que obtendrá estos datos. Nuestra clase `SharedData` es esta:

```
public class SharedData {
    int data;

    public int get()     {
        return data;
    }

    public void put(int newData) {
        data = newData;
    }
}
```

Nuestros hilos `Productor` y `Consumidor` son estos:

```
public class Producer extends Thread {
    SharedData data;

    public Producer(SharedData data) {
        this.data = data;
    }

    @Override
    public void run() {
        for (int i = 0; i < 50; i++) {
            data.put(i);
            System.out.println("Produced number " + i);
            try {
                Thread.sleep(10);
            } catch (Exception e) {}
        }
    }
}
```

```
public class Consumer extends Thread {
    SharedData data;

    public Consumer(SharedData data) {
        this.data = data;
    }

    @Override
    public void run() {
        for (int i = 0; i < 50; i++) {
            int n = data.get();
            System.out.println("Consumed number " + n);
            try {
                Thread.sleep(10);
            } catch (Exception e) {}
        }
    }
}
```

La aplicación principal creará un objeto `SharedData` y un hilo de cada tipo, y iniciará ambos.

```
public static void main(String[] args) {
    SharedData sd = new SharedData();
    Producer p = new Producer(sd);
    Consumer c = new Consumer(sd);
    p.start();
    c.start();
}
```

Si copiamos este ejemplo y vemos cómo funciona, veremos algo como esto:

```
Consumed number 0
Produced number 0
Consumed number 0
```

```
Produced number 1
Consumed number 1
Produced number 2
Produced number 3
Consumed number 3
Produced number 4
Consumed number 4
Consumed number 4
```

Observa cómo, a veces, el productor pone los números demasiado rápido y, a veces, el consumidor también los obtiene demasiado rápido, de modo que no están coordinados (el consumidor puede obtener dos veces el mismo número o el productor puede poner dos números consecutivos).

Podríamos pensar que, si simplemente agregamos la palabra clave `synchronized` a los métodos `get()` y `put()` de la clase `SharedData`, resolveríamos el problema:

```
public class SharedData {
    int data;

    public synchronized int get() {
        return data;
    }

    public synchronized void put(int newData) {
        data = newData;
    }
}
```

Sin embargo, si ejecutamos el programa nuevamente, podemos notar que sigue fallando:

```
Consumed number 0
Produced number 0
Consumed number 1
Produced number 1
Produced number 2
Consumed number 1
Produced number 3
Consumed number 3
Produced number 4
Consumed number 4
```

En realidad, hay 2 problemas que debemos resolver, pero empecemos por el más importante: productor y consumidor tienen que trabajar coordinados; en cuanto el productor pone un número, el consumidor puede obtenerlo, y el productor no podrá producir más números hasta que el consumidor obtenga los anteriores.

Para ello, necesitamos añadir algunos cambios a nuestra clase `SharedData`. En primer lugar, necesitamos un "flag" (bandera) que indique a los productores y consumidores quién es el siguiente en la lista, lo cual dependerá de si hay nuevos datos para consumir (turno para el consumidor) o no (turno para el productor).

```
public class SharedData {
    int data;
    boolean available = false;

    public synchronized int get() {
        available = false;
        return data;
    }

    public synchronized void put(int newData) {
        data = newData;
        available = true;
    }
}
```

Además, debemos asegurarnos de que los métodos `get()` y `put()` se invoquen de forma alternada. Para ello, debemos utilizar el indicador booleano y los métodos `wait()` y `notify()`/`notifyAll()`, de esta manera:

```
public class SharedData {
    int data;
    boolean available = false;

    public synchronized int get() {
        if (!available)
            try {
                wait();
            } catch (Exception e) {}
        available = false;
        notify();
        return data;
    }

    public synchronized void put(int newData) {
        if (available)
            try {
                wait();
            } catch (Exception e) {}
        data = newData;
        available = true;
        notify();
    }
}
```

Mira cómo usamos los métodos `wait()` y `notify()`. Con respecto al método `get()` (llamado por `Consumer`), si no hay nada disponible, esperamos. Luego, obtenemos el número, volvemos a establecer el indicador en false y notificamos al otro hilo. En el método `put` (llamado por `Producer`), si hay algo disponible, esperamos hasta que alguien nos notifique. Luego, establecemos los nuevos datos, volvemos a establecer el indicador en *true* y notificamos al otro hilo.

Si ambos hilos intentan llegar a la sección crítica al mismo tiempo, `el Consumidor` tendrá que esperar (el indicador *disponible* se establece en *falso* al principio) y `el Productor` establecerá

los primeros datos que se consumirán. A partir de ese momento, se alternarán en la sección crítica, consumiendo y produciendo nuevos datos cada vez.

```
Número consumido 0
Número producido 0
Producido el número 1
Número consumido 1
Número consumido 2
Producido el número 2
Producido el número 3
Número consumido 3
Producido el número 4
Número consumido 4
...
```

Ejercicio 7:

Crea un proyecto llamado **Lavavajillas**. Vamos a simular el proceso de lavado de platos en casa; cuando alguien los friega y otro más los seca. Crea las siguientes clases:

- Una clase Plato con solo un atributo entero: el número de plato (para identificar los diferentes platos).
- Una clase PilaPlatos que almacenará hasta 5 platos. Tendrá un método lavar que pondrá un plato en la pila (si hay espacio disponible) y un método secar que cogerá un plato de la pila (si hay alguno). Tal vez necesites un parámetro Plato en el método lavar para añadir un plato a la pila.
- Un hilo Friega que recibirá un número N como parámetro y un objeto de tipo PilaPlatos. En su método run pondrá (lavará) N platos en la pila, con una pausa de 50ms entre cada plato.
- Un hilo Seca que recibirá un número N como parámetro y un objeto de tipo PilaPlatos. En su método run, sacará (secará) N platos de la pila, con una pausa de 100 ms entre cada plato.
- La clase principal creará el objeto PilaPlatos y un hilo de cada tipo (Friega y Seca). Tendrán que lavar/secar 20 platos coordinadamente, por lo que el resultado podría ser algo así (parcialmente):

```
Plato lavado #1, total en pila: 1
Plato secado #1, total en pila: 0
Plato lavado #2, total en pila: 1
Plato secado #2, total en pila: 0
Plato lavado #3, total en pila: 1
Plato lavado #4, total en pila: 2
Plato secado #4, total en pila: 1
Plato lavado #5, total en pila: 2
Plato lavado #6, total en pila: 3
Plato secado #6, total en pila: 2
Plato lavado #7, total en pila: 3
```