2. Programación concurrente

Parte I. Introducción y gestión básica de hilos

Programación de Servicios y Procesos

Nacho Iborra
Álvaro Pérez
Javier Carrasco

Tabla de contenido

1. Introducción a la concurrencia

En esta unidad se repasarán algunos de los conceptos más importantes de la programación concurrente, pero antes de comenzar, deberás familiarizarte con algunos conceptos básicos sobre qué es un programa o proceso, y cómo el sistema operativo los maneja.

1.1. Programas y procesos

Se puede definir un programa como un conjunto de instrucciones empaquetadas como un paquete, que ayuda al usuario a resolver un problema o completar una tarea. Cuando se ejecuta un programa, se convierte en un proceso, esto es, un conjunto de datos e instrucciones colocados en la memoria y manejados por el procesador y el sistema operativo para completar una tarea.

Entonces, cada vez que se ejecuta un programa en el sistema, se está creando un proceso en la memoria para este programa. Este proceso incluye básicamente dos elementos:

- Un contador de programa (*program counter*), un tipo de índice que apunta a la instrucción concreta que se está ejecutando actualmente entre todas las instrucciones que forman parte del programa.
- Un espacio de memoria (*memory space*), donde se colocan, cargan y editan todos los datos e instrucciones que necesita el programa mientras este se ejecuta .

Además, debes tener en cuenta que los procesos son independientes. Si se ejecuta una segunda instancia del mismo programa, se creará otro proceso, con su propio contador de programa y su propio espacio de memoria, y todos los datos y operaciones que realice esta segunda instancia serán completamente independientes de las realizadas por la primera. Puedes verificar esto fácilmente cargando dos veces un navegador web. Puedes cargar diferentes páginas en cada ventana o descargar un archivo con una de ellas mientras lees una página web con la otra. Además, si ejecutas el administrador de tareas de tu sistema operativo, verás dos procesos ejecutándose para el mismo navegador (firefox, chrome o cualquier otro navegador que hayas ejecutado).

1.2. El sistema operativo y sus procesos

El sistema operativo es el intermediario entre el usuario y los programas que utiliza. Ejecuta esos programas e intenta que el usuario se sienta cómodo con la computadora. Además, intenta optimizar el uso de todos los recursos del ordenador (memoria RAM, espacio en disco, uso de CPU, etc).

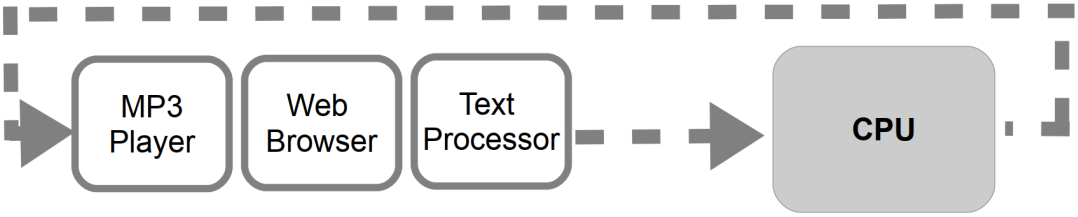
Todos los sistemas operativos modernos pueden ejecutar más de un programa al mismo tiempo. Este es un tipo de simultaneidad que comúnmente se llama multitarea (*multitask*). Gracias a esto, puedes escuchar música mientras escribes un documento o navegas por Internet. Pero este no es el tipo de concurrencia que se tratará en esta unidad.

1.3. Programación concurrente

Como se ha introducido anteriormente, la concurrencia permite que el sistema tenga algunas tareas o procesos ejecutándose al mismo tiempo (la impresión de un documento, un reproductor de música en ejecución, un navegador web descargando archivos ...). Todas estas tareas se pueden ejecutar en:

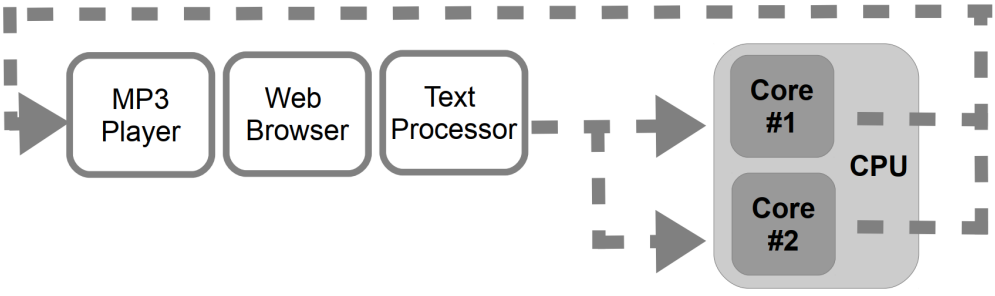
- Un **procesador único**. En este caso, en realidad solo hay un proceso ejecutándose al mismo tiempo, pero el usuario tiene la sensación de tener todas las tareas ejecutándose a la vez.

  Para hacer esto posible, el sistema operativo alterna la ejecución de estos procesos después de un corto período de tiempo (algunos milisegundos). Esto es lo que realmente se llama concurrencia.



Es posible que la concurrencia no mejore el tiempo de ejecución global de todos los programas en ejecución, ya que simplemente se intercambian y esperan un nuevo turno, pero en algunos casos puede ser una mejora en términos de tiempo, especialmente cuando una tarea está ocupando la CPU sin hacer nada (por ejemplo, cuando un programa está esperando que el usuario escriba algo).

- **Múltiples procesadores**, o múltiples núcleos en un procesador. Esto es algo que se está volviendo cada vez más habitual en los procesadores actuales. Suelen tener dos, cuatro o incluso más núcleos, y permiten lo que se llama procesamiento en paralelo. Si se están ejecutando algunos procesos al mismo tiempo, o solo un proceso, y hay suficiente número de núcleos, cada núcleo se encargará de ejecutar uno de estos procesos, o un conjunto dado de instrucciones de un solo proceso, y entonces el rendimiento mejorará significativamente. De lo contrario, si hay más procesos o tareas que núcleos disponibles, entonces el sistema operativo tendrá que aplicar la concurrencia, alternando procesos o conjuntos de instrucciones en uno o algunos núcleos.



En ambos casos (simultaneidad o procesamiento en paralelo), es posible que sea necesario crear un programa que pueda realizar algunas tareas al mismo tiempo. Por ejemplo, un administrador de archivos que permita navegar por el disco duro mientras copia algunos GB de una carpeta a otra. Este es el tipo de técnicas de programación que van a cubrirse en esta unidad.

Hay otros tipos de programas cuyas partes deben ejecutarse por separado o de forma independiente. En este grupo se puede hablar de programación distribuida, donde hay algunas computadoras conectadas en una red, y se necesita ejecutar una parte del programa (típicamente el servidor) en una computadora y la otra parte (típicamente el cliente) en otra computadora diferente. No se verá este tipo de programación en esta unidad, aunque se volverá a este concepto en unidades posteriores.

1.4. Procesos e hilos

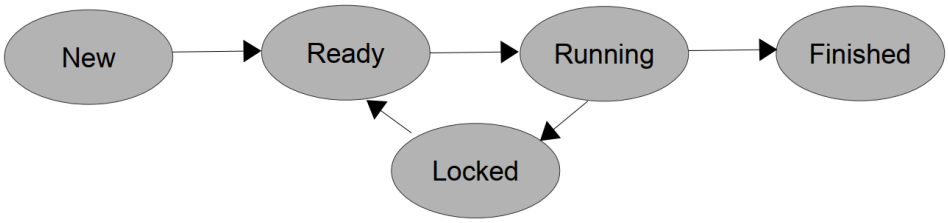
Cuando se habla de programación concurrente, en realidad se habla de múltiples programas que intentan llegar a los mismos recursos (un procesador, un archivo, etc), o un programa que necesita dividir su tarea en pequeñas sub-tareas. En el primer caso, cada uno de estos programas (o instancias del mismo programa) son **procesos** (*processes*). En el segundo caso, cada sub-tarea iniciada dentro del mismo programa es una **hilo** (*thread*). A continuación, se verán las diferencias y similitudes entre estos dos conceptos.

1.4.1. Estados de procesos / hilos

Cada proceso o hilo puede estar en cualquiera de los siguientes estados durante su tiempo de vida:

- **Nuevo** (*New*): el proceso o hilo se acaba de crear, pero aún no ha comenzado a ejecutarse.
- **Listo** (*Ready*): el proceso o hilo no se está ejecutando actualmente, pero está listo para hacerlo.
- **Corriendo** (*Running*): el proceso o hilo se está ejecutando actualmente en un núcleo o procesador determinado.
- **Bloqueado** (*Locked*): el proceso o hilo está esperando que suceda algún evento. Este evento puede ser una entrada del usuario, el desbloqueo de un archivo, etc. Siempre que ocurre este evento, el proceso se pasa a *ready*.
- **Terminado** (*Finished*): el proceso o hilo ha terminado su tarea, o el sistema operativo lo ha obligado a terminar con una interrupción.

El proceso o hilo puede cambiar de un estado a otro en cualquier momento, aunque no todas las combinaciones de cambios son posibles. A continuación, puedes ver un esquema de los estados y sus posibles próximos estados:



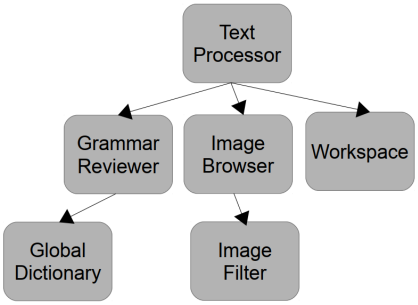
1.4.2. Contexto de proceso / hilo

Cuando un proceso o hilo cambia su estado de *running* a *ready* o *locked* tiene que dejar temporalmente el procesador o núcleo donde se estaba ejecutando, y luego el sistema operativo tiene que guardar todos los datos relacionados con ese proceso, para recuperarlos cuando el proceso regrese al procesador. Estos datos se denominan **contexto** del proceso o hilo, e incluye el estado del proceso y el estado de la memoria (valores de datos, instrucción actual, etc).

Mientras se guarda el contexto, el procesador está esperando, y esto implica que el sistema está perdiendo algo de tiempo mientras tanto. En este punto, hay una muy importante **diferencia** entre procesos e hilos: mientras que los primeros tienen contextos completamente independientes, los hilos que forman parte del mismo proceso padre comparten su espacio de memoria. Por lo tanto, guardar un contexto de hilo es más rápido y fácil que guardar un contexto de proceso, ya que el sistema operativo no tiene que preocuparse por el espacio de memoria, porque será el mismo para todos los hilos. Solo tiene que guardar el índice de la instrucción actual y el estado del hilo. Sin embargo, si no se utilizan los hilos con cuidado, se pueden alterar los datos compartidos que no debían alterarse. Dependiendo de los datos que se estén manejando, la velocidad esperada y el lenguaje de programación usado, se utilizarán procesos o hilos para implementar un programa concurrente.

1.4.3. Árboles de procesos / hilos

Llegados a este punto, ya deberías saber que, siempre que se ejecuta un programa, se crea un proceso en memoria. Luego, a partir de este proceso inicial, se pueden crear y lanzar otros procesos o hilos, dependiendo de la aplicación. Y posteriormente, se pueden lanzar más procesos o hilos de los anteriores, etc. Esto crea un árbol enraizado en el programa o aplicación inicial, en el que cada sub-proceso o hilo puede ser el padre de otros procesos o hilos. Por ejemplo, si ejecutas un procesador de texto, es posible que tengas todos estos procesos o hilos ejecutándose al mismo tiempo:



1.4.4. Conclusión

Se acaban de ver algunos conceptos relacionados con procesos e hilos. En este punto deberías ver que ambos tienen algunos aspectos en común (estados, contexto, árbol), pero también tienen algunas diferencias en cuanto al tamaño del contexto y la memoria compartida que pueden ser cruciales cuando se debe tomar una decisión sobre si usar procesos o hilos.

1.5. Etapas de la programación concurrente

Ya sea si utilizas procesos o hilos, debes seguir algunos pasos comunes si quieres que el programa esté dividido o estructurado de manera óptima para resolver un problema de concurrencia de manera eficiente:

1. En primer lugar, tienes que dividir la funcionalidad del programa en partes o módulos más simples.
2. Luego, distribuye esas partes en diferentes procesos / hilos, y establece el esquema de comunicación entre ellos, para que puedan enviarse y recibir datos entre sí. Cuando se planifica la distribución, se debe tener en cuenta maximizar la independencia de cada proceso / hilo (o minimizar las comunicaciones entre ellos).
3. Finalmente, es el momento de implementar los procesos / hilos.

En las siguientes secciones, se explicará cómo usar procesos e hilos en uno de los lenguajes de programación más populares en la actualidad: Java. La unidad terminará con algunos aspectos avanzados de la programación concurrente en Java, como nuevos *frameworks* para hacer aplicaciones más eficientes en sistemas con múltiples núcleos, o técnicas de programación para manejar colecciones de elementos de manera concurrente.

2. Gestión de hilos en Java

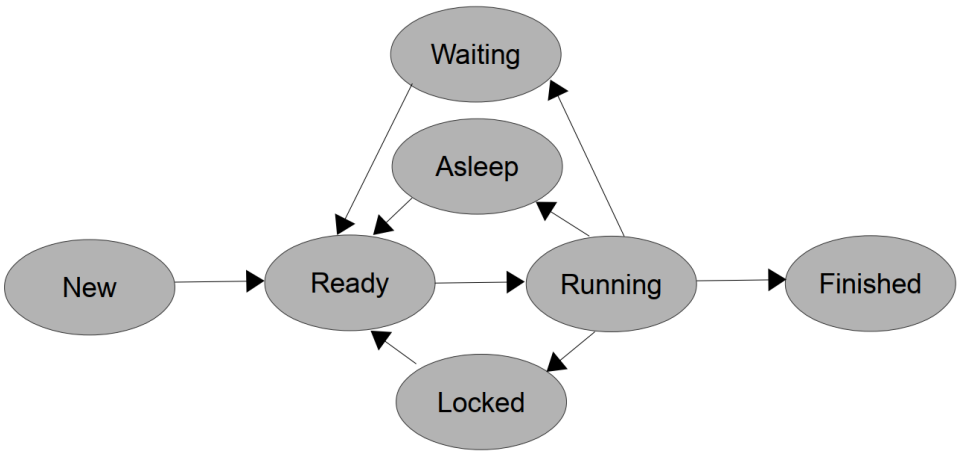
Al hablar de Java se debe tener en cuenta que todo es un hilo, incluso la aplicación principal, y todo lo que se genera desde esta aplicación es un hilo, por lo que el elemento más importante de la programación concurrente en Java son hilos. Como se ha visto antes, un hilo es una especie de sub-proceso o sub-tarea cuyo contexto se comparte parcialmente con el resto de hilos de la misma aplicación. Para ser más precisos, todos los hilos de la misma aplicación tienen el mismo espacio de memoria, por lo que todos comparten los mismos datos.

Se pueden hacer más o menos el mismo tipo de operaciones con hilos y procesos: crearlos, sincronizarlos, destruirlos, etc. Pero notarás en unos minutos que el “mundo hilo” ofrece un amplio abanico de posibilidades que no puedes encontrar para los procesos. Esto se debe a que su aplicación principal de Java ya es un hilo y, por tanto, Java se centra en los hilos.

2.1. Estados del hilo

Los procesos e hilos pasan por los mismos estados a lo largo de su tiempo de vida, como se ha explicado anteriormente (ver sección 1.4. Procesos e hilos). Pero, además de esos estados, se podrían añadir algunos más a esa lista, con respecto a los hilos de Java:

- **Dormido** (*Asleep*): el hilo se ha quedado dormido debido a una llamada al método *sleep()* que se verá más adelante. Tan pronto como expire el tiempo de reposo, volverá al estado *Ready*.
- **Esperando** (*Waiting*): el hilo está esperando que otro hilo lo reactive. Ocurre cuando los hilos están peleando por recursos limitados, y quien los obtiene es el encargado de avisar a los demás cuando ha terminado. También se verá esta característica más adelante.
- Un hilo puede llegar a estos estados desde el estado *Running* (solo cuando está en ejecución se le puede pedir que duerma o espere), y cuando se despierta, vuelve al estado *Ready* hasta que se ejecute de nuevo. Entonces, con estos nuevos estados, el esquema para los estados de los hilos de Java sería así:



Existen otros estados que han quedado obsoletos, como *Suspended* (en versiones anteriores de Java, se podían pausar y reanudar hilos desde cualquier lugar, lo que era potencialmente peligroso para la integridad de los datos y las aplicaciones) o *Stopped* (una forma de forzar el final de un hilo, que también era potencialmente peligrosa).

2.2. Manejo básico de hilos. Crear y lanzar hilos

2.2.1. Definiendo un hilo

Si quieres definir un hilo, dispones de algunas formas de hacerlo:

1. Hay una clase en Java llamada ***Thread***, que se puede utilizar para crear hilos heredando de ella, e implementar (*override*) su método *run()*.

```
public class MyThread extends Thread {
    … // Attributes, constructors and methods of our class

    @Override
    public void run() {
        // Code to be executed by the thread
    }
}
```

2. También se puede crear una clase que implemente la interfaz ***Runnable*** e implementar su método *run()*.

```
public class MyOtherThread implements Runnable {
    … // Attributes, constructors and methods of our class

    @Override
    public void run() {
        // Code to be executed by the thread
    }
}
```

   En este último caso, también se puede usar una clase anónima o una expresión lambda para definir el objeto *Runnable*.

```
Runnable lambdaRun = () -> {
    // Code to be executed by the thread
};
```

Como puedes ver, en todos estos casos, se necesita definir (*override*) el método *run()* que "hereda" de la clase *Thread* o de la interfaz *Runnable*. Esta será el principal método del hilo.

2.2.2. Crear y lanzar un hilo

Para lanzar un hilo (recuerda, la aplicación principal también es un hilo que se ejecuta en la JVM), no hay que llamar al método *run()* directamente: no habría ninguna multitarea, ya que el hilo actual (normalmente la aplicación principal) ejecutaría el método *run()*, pero no crearía un hilo en sí. En lugar de esto, habrá que llamar al método *start()* que tiene cada hilo, y luego el sistema carga el estado del hilo en la memoria y llama al *run()* indicado, de modo que se puedan tener tantos hilos como se necesiten, ejecutándose todos juntos.

Si se define un hilo extendiendo de la clase ***Thread***, después se puede crear un objeto hilo y ejecutarlo con estas instrucciones (de acuerdo con el ejemplo anterior):

```
Thread t = new MyThread();
t.start();
```

Si se define el hilo implementando la interfaz ***Runnable***, entonces se puede crear y ejecutar un hilo definiendo una nueva instancia de *Thread* con un objeto *Runnable* como parámetro. Observa ambos ejemplos (clase normal y expresión lambda) creados antes:

```
// Normal class that implements Runnable
Thread t = new Thread(new MyOtherThread());
t.start();

// Lambda expression
Thread t = new Thread( lambdaRun);
t.start();
```

Al hacer esto, el objeto *Thread* que se acaba de crear sabe dónde encontrar su método *run()*: en el objeto *Runnable* que recibe como parámetro.

2.2.3. ¿Extender *Thread* o implementar *Runnable*?

Como te encontrarás en muchas otras situaciones a lo largo de tu carrera como programador, existen diferentes formas de hacer lo mismo. En este caso, se puede crear y lanzar un hilo en dos sabores: extendiendo de la clase *Thread* o implementando la interfaz *Runnable*. Al final, el comportamiento del hilo creado será el mismo, pero existen algunas diferencias o motivos para elegir una forma y no la otra:

- Si extiendes de la clase *Thread*, no podrás heredar de ninguna otra clase. Por tanto, utiliza esta forma solo cuando tu clase hilo no necesite heredar de nada más. Esta opción es habitual en aplicaciones pequeñas y sencillas.
- Por el contrario, tienes la “opción B”, es decir, implementar la interfaz *Runnable* (o usando clases anónimas o expresiones lambda). Recuerda, puedes implementar múltiples interfaces, pero solo puedes extender una clase. Es por eso que Java deja esta puerta abierta: en caso de que ya hayas extendido otra clase, aún puedes aplicar hilos en ella. Esta opción es más habitual en aplicaciones complejas.

2.2.4. Ejemplo

Observa el siguiente ejemplo para ver cómo funciona un hilo. Para empezar con algo simple, se creará un hilo que cuente del 1 al 10. Como no es necesario extender desde ninguna otra clase, se creará una subclase de *Thread*. En ejemplos posteriores se utilizará la interfaz *Runnable*, para que veas cómo trabajar con ambas opciones.

El hilo básico quedará como se muestra a continuación:

```
public class MyCounterThread extends Thread {
    @Override
    public void run() {
        for (int i = 1; i <= 10; i++) {
            System.out.println("Counting " + i);
        }
    }
}
```

El programa principal, que se encargará de crear y lanzar los hilos quedará así:

```
public class MyMainCounter {
    public static void main(String[] args) {
        MyCounterThread t = new MyCounterThread();
        t.start();
    }
}
```

Copia estas clases en un proyecto y ejecuta el programa principal para ver que funciona correctamente. Ahora, se añadirán algunos cambios al programa principal para ver cómo cambia su comportamiento inicial. Si se añade esta línea al final del método *main()*:

```
public class MyMainCounter {
    public static void main(String[] args) {
        MyCounterThread t = new MyCounterThread();
        t.start();
        System.exit(0);
    }
}
```

¿Qué pasa cuando vuelves a ejecutar el programa? Si ejecutas el programa varias veces, descubrirás que a veces cuenta hasta 10, a veces no cuenta nada… y a veces cuenta entre 1 y 10. Esto se debe a que el programa principal finaliza inesperadamente con el método *exit()*, y luego todos sus hilos también se eliminan. Si el hilo comenzó a ejecutarse antes de que se elimine su padre, podrá contar algunos números.

Ahora cambia esa instrucción por esta:

```
public class MyMainCounter {
    public static void main(String[] args) {
        MyCounterThread t = new MyCounterThread();
        t.start();
        System.out.println("Hello!!");
    }
}
```

¿Qué pasa ahora? El hilo cuenta hasta 10, y en algún punto intermedio de la cuenta aparece el mensaje *"Hello!!"*. Quizás se muestre antes del número 1, o después del número 7 ... Dependerá del momento en que el programa principal llegue al procesador para imprimir su mensaje.

Finalmente, intenta llamar al método *start()* nuevamente después de su primera llamada:

```
public class MyMainCounter {
    public static void main(String[] args) {
        MyCounterThread t = new MyCounterThread();
        t.start();
        t.start();
    }
}
```

Verás que se lanza una excepción de tipo *IllegalThreadStateException*. No se puede llamar al método *start()* **más de una vez**. Se tendrá que crear un nuevo objeto *Thread*.

2.2.5. Conclusiones

De este ejemplo, se pueden sacar algunas conclusiones:

- Cuando se lanza un hilo desde la aplicación principal, comienza su ejecución independiente y paralelamente.
- Cuando la aplicación principal finaliza correctamente, el hilo sigue ejecutando su tarea hasta que finaliza.
- Cuando la aplicación principal se ve obligada a finalizar, el hilo también termina inesperadamente. Para ser más precisos, si algún hilo de la aplicación llama al método *System.exit()*, todos los hilos terminarán su ejecución.
- No hay forma de saber el orden exacto en el que la aplicación principal y sus hilos producirán sus resultados. Dependerá del planificador de tareas. De todos modos, se aprenderá a dar más tiempo de CPU a algunos hilos a expensas de los otros en breve, y también a sincronizar o coordinar hilos para producir resultados en un orden determinado.
- Después de lanzar un hilo, no se podrá llamar a su método *start()* de nuevo. Pero se podrá llamar a algunos otros métodos para obtener su estado y algunas otras características, esto se verá más adelante.

Ejercicio 1

Crea un proyecto llamado ***FibonacciThread***. Define una subclase de *Thread* que muestre números de *Fibonacci* hasta un parámetro N dado que se pasará al constructor.

Los números de *Fibonacci* son una secuencia que comienza por 1 y 1, en la que cada nuevo número se calcula sumando los dos números anteriores de la secuencia. La secuencia sería la siguiente: 1, 1, 2, 3, 5, 8, 13, 21…

Ejercicio 2

Crea un proyecto llamado ***MultiplierThreads***. Define una subclase de *Threads* que tenga un número como atributo. Asigne un valor a este número a través del constructor de la clase. En el método *run()*, el hilo tiene que mostrar la tabla de multiplicar de su atributo.

Después, desde la aplicación principal, crea 10 hilos (cada uno con un número diferente) y ejecútalos al mismo tiempo. Verás cómo los mensajes de un hilo se mezclan con los mensajes de otros hilos. Por ejemplo…

```
1 x 0 = 0
1 x 1 = 1
3 x 0 = 0
4 x 0 = 0
…
```

2.3. Información básica del hilo

Hay algunos métodos y propiedades útiles en la clase *Thread* para obtener y establecer información sobre un hilo. Se centrará la atención en los tres siguientes por ahora:

- Cómo configurar y obtener el nombre del hilo.
- Cómo obtener el estado del hilo.
- Cómo obtener el identificador del hilo.

2.3.1. Configurar y obtener el nombre del hilo

Si deseas dar un nombre a tus hilos, simplemente puedes añadir un atributo nombre a tu clase (ya sea extendiendo *Thread* o implementando *Runnable*). Pero hay algunos métodos en la clase *Thread* que permite establecer y obtener este nombre sin añadir ninguna información adicional: el método ***setName()*** establece el nombre del hilo, el método ***getName()*** obtendrá este nombre.

```
Thread t = new MyCounterThread();
t.setName("MyThread A");
t.start();
System.out.println("Thread " + t.getName() + " has been launched");
```

En este ejemplo, se ha creado un hilo, estableciendo su nombre y luego imprimiéndolo unas líneas más abajo. Si quieres obtener/establecer el nombre del hilo dentro del propio hilo (por ejemplo, desde el método *run()* del hilo), se puede llamar al método ***currentThread()*** para obtener un objeto *Thread* que apunta al hilo actual, y luego obtener/establecer su nombre.

```
@Override
public void run() {
    Thread.currentThread().setName("AAA");
    …
    System.out.println(Thread.currentThread().getName());
}
```

Si ejecutas un hilo desde una instancia *Runnable*, por ejemplo, puedes establecer el nombre directamente al crear el hilo como segundo parámetro en el constructor.

```
Runnable counterRun = () -> {
    System.out.println(Thread.currentThread().getName() + " running");
    for (int i = 1; i <= 10; i++) {
        System.out.println("Counting " + i);
    }
};
Thread t = new Thread(counterRun, "CounterThread");
t.start();
```

2.3.2. Obtener el estado del hilo

También se puede obtener el estado actual del hilo en cualquier momento. Para administrar estos estados, hay una enumeración interna llamada ***Thread.State***, y un método ***getState()*** en la clase *Thread*. El siguiente ejemplo lanza un hilo y, unas pocas líneas a continuación, verifica su estado actual:

```
Thread t = new MyCounterThread();
t.start();
…
Thread.State st = t.getState();
```

El valor devuelto del método *getState()* puede ser uno de los siguientes estados, que están representados por constantes en el *enum Thread.State*: *NEW*, *RUNNABLE*, *BLOCKED*, *WAITING*, *TIMED WAITING* o *TERMINATED*. Por ejemplo, si quieres comprobar si el hilo ha finalizado su tarea, se podría hacerlo así:

```
if (st == Thread.State.TERMINATED)
    System.out.println("Thread is terminated.");
```

También se puede comprobar si un hilo ha terminado su tarea con el método ***isAlive()*** (de la clase *Thread*):

```
if (!t.isAlive())
    System.out.println("Thread is terminated.");
```

2.3.3. Obtener el identificador del hilo

Java Virtual Machine asigna un identificador único a cada hilo creado. Si quieres conseguirlo, solo tienes que llamar al método ***getId()*** de la clase *Thread*:

```
@Override
public void run() {
    …
    System.out.println("Thread #" + Thread.currentThread().getId());
}
```

2.4. Los métodos de sueño y rendimiento

En esta sección se aprenderá como poner en reposo los hilos, o pedirles que dejen libre el procesador.

2.4.1. Método *sleep*

Cuando se llama al método ***sleep()***, el hilo que lo está llamando se duerme automáticamente (es decir, detiene su ejecución), hasta que expira el número de milisegundos indicado en el parámetro. Esto es útil para dejar que el procesador esté libre para otros hilos, si el hilo actual no tiene nada que hacer por ahora, o si se pretende ayudar a mejorar la concurrencia entre nuestros hilos.

El método *sleep()* es un método estático de la clase *Thread*, por lo que para llamarlo solo tienes que añadir esta instrucción en la posición donde quieras que duerma el hilo, con el tiempo expresado en milisegundos:

```
Thread.sleep(2000);
```

Este ejemplo pone el hilo que ejecuta la instrucción en reposo durante 2 segundos (2000 milisegundos). De hecho, se necesita capturar una posible excepción que se puede generar al usar este método.

```
try {
    Thread.sleep(2000);
} catch (InterruptedException e) {
    …
}
```

Debes tener en cuenta que, incluso si utilizas un objeto *Thread* para llamar a este método …

```
public static void main(String[] args) {
    Thread t = new MyThread();
    t.start();
    t.sleep(2000);
}
```

El hilo representado por el objeto *t* no dormirá, pero la aplicación principal sí lo hará. Recuerda: el hilo que llama al método es el que duerme.

En cuanto a los milisegundos, también se puede utilizar la case ***TimeUnit*** (del paquete ***java.util.concurrent***) y sus propiedades para especificar otra unidad de tiempo, que se convertirá automáticamente a milisegundos. Por ejemplo, si quieres que el hilo duerma 5 segundos, también puedes hacerlo así:

```
import java.util.concurrent.TimeUnit;
…
try {
    TimeUnit.SECONDS.sleep(5);
} catch (InterruptedException e) {
    …
}
```

Puedes echar un vistazo a la API de Java para ver más constantes que puedes usar desde la clase *TimeUnit*, como *MINUTES*, *HOURS*, etc. La llamada a *TimeUnit.sleep()* genera en realidad una llamada *Thread.sleep()*, con la conversión apropiada a milisegundos.

2.4.2. Método *yield*

El método ***yield()*** es similar al método *sleep()*, pero no necesita una cantidad de milisegundos como parámetro. Simplemente deja el procesador libre para que el planificador de tareas pueda asignarlo a otro hilo. Si ningún otro hilo está esperando al procesador, entonces el hilo que produjo la llamada vuelve al procesador.

Este método también es estático y también se aplica al hilo que lo llama. No lanza ninguna excepción cuando se llama, por lo que puede usarse simplemente así:

```
Thread.yield();
```

Existe un problema potencial al usar el método *yield()*: el planificador de tareas de JVM puede ignorar esta instrucción, por lo que no es posible estar seguros de que un hilo cederá cuando se le pida.

2.4.3. Ejemplo

En este ejemplo, se definirá un hilo (implementando la interfaz *Runnable* a través de una expresión lambda) que cuenta de la A a la Z, durmiendo 100 ms después de imprimir cada letra. El programa principal esperará a que termine este hilo, comprobando su estado después de cada iteración.

```
public static void main(String[] args) {
    Thread t = new Thread(() -> {
        for (char c = 'A'; c <= 'Z'; c++) {
            System.out.println(c);
            try {
                Thread.sleep(100);
            } catch (InterruptedException e) {
                System.err.println("Error: Thread interrupted");
            }
        }
    });

    t.start();

    do {
        try {
            Thread.sleep(100);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    } while (t.isAlive());

    System.out.println("Thread has finished, and so do I");
}
```

Observa que el programa principal solo duerme unos milisegundos (pueden ser 50, 100, 200… no importa) en cada iteración. Solo tiene que esperar a que termine el hilo, no tiene que hacer nada, así que mejor dejar libre el procesador *sleeping* o *yielding*.

Ejercicio 3

Crea un proyecto llamado ***ThreadRace***. Define una subclase de *Thread* y crea 3 objetos de esta subclase. Cada hilo tendrá su propio nombre A, B y C, y deberán contar de 1 a 1000. El programa principal tendrá que esperar a que finalicen todos sus hilos, y deberá dormir 100 ms después de cada iteración y mostrar en pantalla el recuento actual de cada hilo. Por ejemplo:

```
…
Thread A: 96     Thread B: 112    Thread C: 105
Thread A: 116    Thread B: 134    Thread C: 121
…
```

También puedes llamar al método *sleep()* en cada iteración del método *run()* si consideras que sus hilos se están ejecutando demasiado rápido. Simplemente añade esta línea dentro de su bucle y el conteo se ralentizará:

```
this.sleep((int) (Math.random() * 10));
```

2.5. Terminar e interrumpir hilos

Hay dos formas de forzar a un hilo a terminar su tarea: utilizar *flags* booleanos para decirle al hilo que debe detenerse cuando verifique esos *flags*, o utilizar interrupciones para detenerlo.

2.5.1. Terminar hilos con *flags*

Los hilos terminan su tarea cuando ejecutan cada instrucción de su método *run()*. No hay forma de poder detener un hilo en un momento dado (estaban los métodos un *stop()* y *destroy()* en versiones anteriores de Java, pero ahora están en desuso). Incluso si se ponen sus variables a nulo, los recursos del hilo se mantendrán bloqueados.

Pero no te preocupes. Todavía existe un método para pedirle a un hilo que termine, aunque no termine en ese preciso momento. Este método se aplica a hilos que tienen algún tipo de bucle en su método *run()*. Si se implementa el bucle correctamente, se puede utilizar un *flag* para decirle al hilo si puede continuar o si debe terminar.

Fíjate en este método con un ejemplo. Si se define una subclase de hilo como la siguiente:

```
public class KillableThread extends Thread {
    boolean finish = false;

    public void setFinish(boolean finish){
        this.finish = finish;
    }

    @Override
    public void run() {
        while (!finish){
            … // Thread task
        }
    }
}
```

Luego se puede crear y lanzar un hilo desde la aplicación principal, y pedirle al hilo que termine con su método *setFinish()*:

```
public static void main(String[] args) {
    KillableThread kt = new KillableThread();
    kt.start();
    ...
    if (someCondition) {
        kt.setFinish(true);
    }
}
```

Tan pronto como el hilo llegue al inicio del bucle y compruebe que la variable *finish* es cierto, terminará el método *run()*.

Ejercicio 4

Crea un proyecto llamado ***ThreadRaceKilled*** basado en el proyecto creado en Ejercicio 3.

Modifica la aplicación principal para que, tan pronto como el hilo A llegue a 700, se le pida que finalice (con una variable booleana). Siéntate libre de añadir todo el código que necesites a cada clase del proyecto.

2.5.2. Terminar hilos con interrupciones

Hay una segunda forma de terminar un hilo. Consiste en llamar al método *interrupt()* del hilo. Observa esto en el siguiente ejemplo:

```
public static void main(String[] args) {
    Thread t = new Thread(() -> {
        try {
            while (!Thread.currentThread().isInterrupted()) {
                System.out.println("Running");
                Thread.sleep(100);
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("Finished by an interruption");
    });

    t.start();
    try {
        // Wait for a while...
        Thread.sleep(100);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    t.interrupt();
}
```

En este ejemplo, se crea un hilo que comprueba en cada bucle si ha sido interrumpido (con el método *isInterrupted()*, de la clase *Thread*). De lo contrario, sigue funcionando (es decir, imprime un mensaje y duerme 10 ms). Desde el hilo principal, se espera unos milisegundos y luego se interrumpe el hilo creado previamente con el método *interrupt()*. Este método provoca una *InterruptedException* que hace que el hilo vaya a la sección *catch* y el método *run()* termina.

La *InterruptedException* solo se lanza por la llamar a *sleep()* en el *Runnable*. Si no se llama a *sleep()*, *wait()*, *join()* o cualquier otro método que pueda lanzar esta excepción, la estructura *try ... catch* no sería necesaria, y este hilo terminaría llamando a su método *isInterrupted()*. Observa que un hilo decide si responde a la interrupción o no, utilizando su método *isInterrupted()* y/o capturando las posibles excepciones que se pueden lanzar.

2.6. Hilos, contexto y datos compartidos

Cada hilo creado por la misma aplicación comparte un contexto común. ¿Qué significa esto exactamente? Echa un vistazo al siguiente ejemplo (algunas líneas están numeradas para explicarse más adelante):

```
public class ContextExample implements Runnable {
    // Reference to current thread
    Thread t;

    public void start2Threads() {
        // Create first thread
        t = new Thread(this);
        t.start();

        // Sleep for 5 seconds
        try {
            Thread.sleep(5000);
        } catch (InterruptedException e) { }

        // Create second thread
        t = new Thread(this);                                    // Line #1
        t.start();

        // Sleep for 5 seconds
        try {
            Thread.sleep(5000);
        } catch (InterruptedException e) { }

        // Destroy thread
        t = null;                                                // Line #2
    }

    @Override
    public void run() {
        // Take initial time in milliseconds
        long ini = System.currentTimeMillis();

        while (t == Thread.currentThread()) {
            System.out.println("Running thread (" + ini + ")");
            // Sleep for 100 ms
            try {
                Thread.sleep(100);
            } catch (InterruptedException e) { }
            System.out.println("Finishing thread (" + ini + ")");
        }
    }

    public static void main(String[] args) {
        ContextExample t = new ContextExample();
        t.start2Threads();
    }
}
```

Escribe o copia este código en un proyecto. Pruébalo e intenta responder las siguientes preguntas antes de leer las respuestas:

1. ¿Qué hace la condición del *while* en el método *run()*?

   Sigue en dentro del bucle mientras la variable *t* apunta al hilo que se está ejecutando actualmente. Cuando esta variable apunta a otro hilo (sucede en la línea 1), entonces el hilo anterior termina su tiempo en el *while*.

2. ¿Puede haber dos hilos ejecutando sus métodos *run()* al mismo tiempo?

   Sí. Tan pronto como la línea 1 se ejecuta, el segundo hilo está listo para comenzar. Puede suceder que se inicie antes de que el hilo anterior compruebe su condición del *while* o termine su método *run()*. En tal caso, ambos hilos estarían ejecutando sus métodos *run()*.

3. Si la respuesta a la pregunta anterior es sí, ¿Podrían esos hilos entrar en conflicto con variables *ini*, por lo que un hilo sobrescribiría el valor escrito previamente por el otro?

   No, la variable *ini* es una variable local del método *run()*, por lo que cada llamada a dicho método crea su propia variable local *ini*. Sin embargo, el atributo *t* se comparte para todos los hilos creados. Es por eso que, cuando cambia su valor, el hilo anterior finaliza su tarea y el nuevo hilo se asocia a esa variable (ambos hilos comparten el mismo valor para *t*).

4. ¿Cómo se puede detener un hilo en este ejemplo sin crear uno nuevo?

   Solo habría que poner el atributo t a nulo, como en la línea 2.

2.6.1. Conclusiones

Después de probar este ejemplo, se puede llegar a algunas conclusiones:

1. Cada atributo del mismo objeto se comparte entre todos los hilos de la aplicación. Por eso, cuando el programa principal cambia el valor del atributo *t* en el ejemplo anterior, ambos hilos ven ese cambio.

2. Si se crea una subclase de *Thread* en lugar de implementar la interfaz *Runnable*, cada atributo de esta subclase no se comparte entre hilos, ya que se instancia en cada hilo y así crea su propio espacio de memoria, como se hace con cada objeto instanciado. Por ejemplo, si se define esta clase:

```
public class MyThread extends Thread {
    int num;

    public MyThread(int num) {
        this.num = num;
    }
    …
}
```

Así, el atributo *num* será diferente para cada hilo instanciado. Por lo que, si se escribe algo como esto:

```
MyThread t1 = new MyThread(10);
MyThread t2 = new MyThread(20);
```

Entonces, el objeto *t1* tendrá su atributo *num* con valor 10, y *t2* lo tendrá con valor 20.

3. Si se llama a un método varias veces (por ejemplo, al método *run()* cada vez que se crea e inicia un hilo), sus variables locales serán diferentes en cada llamada (no se comparten). Es por eso que cada hilo en el ejemplo anterior tiene su propio valor *ini*.

2.7. Sincronización o coordinación de hilos

Existen diferentes formas de sincronizar, o coordinar, hilos cuando se inician desde la misma aplicación. La forma más sencilla de coordinación es unir hilos, es decir, hacer que un hilo espere hasta que otro hilo termine su tarea por completo. A partir de ese momento, existen estructuras de sincronización más complejas, como exclusión mutua, bloqueos, etc. A continuación, se verán algunas de estas técnicas en este apartado.

2.7.1. Coordinación básica. Unir hilos

Si se quiere que un hilo espere hasta que termine otro hilo, se puede usar el método *join()* del hilo que debe esperar. En este ejemplo, la aplicación principal crea un hilo y espera hasta que termine antes de continuar:

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

De hecho, el método *join()* puede lanzar una *InterruptedException*, por lo que se tendrá que capturar:

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

Si quieres que un hilo secundario (no un programa principal) espere a otro hilo, entonces necesitas decirle a este hilo cuál es el hilo que debe esperar. Normalmente se utilizará un atributo dentro de la clase del hilo para almacenar esta información:

```
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) {
            try {
                waitThread.join();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

Luego, en la aplicación principal, se crean dos hilos de tipo *MyThread*, y se le pide a uno de ellos que espera al otro:

```
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 5

Crea un proyecto llamado ***ThreadRaceJoin*** basado en 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 haya finalizado el hilo anterior. Entonces, el hilo A comenzará al principio del programa, el hilo B comenzará cuando termine el hilo A y el hilo C comenzará cuando termine el hilo B. El programa principal esperará hasta que el último hilo (C) termine la carrera.

Ejercicio 6

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

```
1 x 0 = 0
1 x 1 = 1
…
1 x 10 = 10
2 x 0 = 0
…
```

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

Es bastante habitual que varios hilos quieran obtener el mismo recurso (por ejemplo, una variable, un archivo de texto, una base de datos, etc), 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 lo está usando).

El fragmento de código que está a cargo 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 al mismo tiempo. Para lograr esto, Java ofrece algunas opciones.

Observa el problema en profundidad con el siguiente ejemplo: en primer lugar, se creará un objeto de clase *Counter*, que se compartirá entre todos los hilos:

```
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 un solo atributo, *value*, que es el valor que será leído y/o modificado por los hilos, llamando a los métodos *increment()* o *decrement()*.

Luego, se crean dos tipos de hilos: uno que incrementará *value* de *Counter* 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 se ejecuta el ejemplo. A veces será 105, otras veces 97, etc, pero siempre debería ser 100 (se espera que un hilo incremente el valor 100 veces y se espera que el otro hilo lo reduzca 100 veces también).

¿Por qué puede pasar esto? Bueno, puede ocurrir que *tinc* se meta en el método *increment()* y luego el control va a *tdec*, que se mete 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 pasa a *tdec* que lee el mismo valor 100 (*tinc* aún no lo ha cambiado) y lo establece en 99, luego, cuando el control vuelva a tinc, establecerá el valor en 101 y la disminución habrá desaparecido.

Para resolver 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 mecanismo de sincronización suspende el hilo que intenta entrar en la sección crítica. Si hay más de un hilo esperando que otro hilo termine la sección crítica, tan pronto como finalice, la JVM elige uno de los hilos en espera (aleatoriamente) para ejecutarlo. Observa cómo funciona este mecanismo y sus variantes.

2.7.3. Métodos de sincronización

Uno de los métodos más básicos de sincronización en Java es la palabra clave *synchronized*. Se puede utilizar para controlar el acceso a un método, de modo que se convierta en una sección crítica.

Java solo 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 solo permite la ejecución de una sección crítica por objeto y una sección crítica estática por clase.

En el ejemplo anterior, si solo se añade la palabra clave *synchronized* para 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 vuelves a ejecutar el programa, notarás 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 habrá problemas en incrementar y decrementar el valor al mismo tiempo, ya que ambos hilos están compartiendo el mismo objeto *Counter*, y solo un hilo puede ejecutar un método *synchronized* al mismo tiempo.

También notarás que el programa se ejecuta más lento que antes. Este es uno de los efectos de la sincronización, penaliza el rendimiento de la aplicación.

Ejercicio 7

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. Añade un constructor para inicializar el dinero en la cuenta y los métodos *addMoney()* y *takeOutMoney()*, que sumará o restará la cantidad pasada como parámetro. Añade el método *getBalance()* también, 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 de la clase *Thread* o implementar la interfaz *Runnable* para hacer esta clase. En el método *run()*, el hilo añadirá 100€ a la cuenta bancaria 5 veces, durmiendo 100 ms entre cada operación.
- Una clase *BankThreadSpend* con un atributo de tipo *BankAccount*. Puedes extender de la clase *Thread* o implementar la interfaz *Runnable* para hacer esta clase. En el método *run()*, el hilo sacará 100€ de la cuenta bancaria 5 veces, durmiendo 100 ms entre cada operación.
- Desde la clase principal, crea un objeto *BankAccount* y un *array* con 20 objetos *BankThreadSave* y 20 BankThreadSpend, utilizando todos ellos el mismo objeto *BankAccount*. Inicializa todos y observa cómo cambia el saldo de la cuenta bancaria (imprime un mensaje en algún punto para mostrar el saldo después de cada operación).

- En este punto, deberías haber notado que la cuenta bancaria no funciona correctamente. Añade los mecanismos de sincronización que consideres para solucionar el problema.

2.7.4. Sincronizar objetos

También se puede 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 intente ejecutar las instrucciones dentro de este bloque, no podrá hacerlo si otro hilo B ya está ejecutando una sección crítica que afecta al objeto. Tan pronto como el hilo B termine con la sección crítica, el otro hilo A se despertará y entrará en la sección crítica.

Por supuesto, se puede usar cualquier otro objeto con la palabra clave *synchronized*. Por ejemplo, si tienes un objeto llamado *file* y quieres crear una sección crítica a su alrededor, podemos hacerlo así:

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

Ejercicio 8

Crea un proyecto ***BankAccountSynchronizedObject*** basado en ejercicio anterior. En este caso, no puedes sincronizar ningún método, solo puedes sincronizar objetos. ¿Qué cambios añadirías al proyecto para asegurarte que seguirá funcionando correctamente?