UNIDAD 2

PROGRAMACIÓN MULTIHILO

1 INTRODUCCIÓN A LA PROGRAMACIÓN MULTIHILO

Dependiendo del número de flujos de ejecución que usan los programas para realizar sus tareas, podemos hablar de dos tipos de programas:

- Programa de flujo único: realiza todas sus tareas en un único flujo de ejecución. Esto implica que para realizar múltiples tareas deberá hacerlo de forma secuencial, es decir, cada tarea que inicie debe concluirla por completo antes de que pueda iniciar la siguiente.
- Programa de flujo múltiple: realiza cada tarea en su propio flujo de ejecución. Cada flujo de ejecución se inicia y finaliza de forma independiente, pudiéndose ejecutar todos ellos concurrentemente.

La programación multihilo, multithreading en inglés, consiste en desarrollar programas o aplicaciones de flujo múltiple, en los que cada flujo de ejecución recibe el nombre de hilo.

Formalmente, un hilo o thread, también conocido como proceso ligero, es una secuencia de código en ejecución dentro del contexto de un proceso padre, lo que implica que:

- No pueden ejecutarse de forma autónoma, necesitan la supervisión del proceso padre.
- Dentro de cada proceso puede haber varios hilos ejecutándose.
- Dentro de un mismo proceso, los hilos comparten datos, código y espacios de direcciones.

Ventajas de los hilos sobre los procesos:

- Se tarda mucho menos tiempo en crear un nuevo hilo dentro de un proceso existente que en crear un nuevo proceso.
- Se tarda mucho menos tiempo en terminar un hilo que un proceso.
- Se tarda mucho menos tiempo en conmutar entre hilos de un mismo proceso que entre procesos.
- La comunicación entre hilos es más rápida debido a que, al compartir memoria y recursos, se pueden comunicar entre sí sin invocar el núcleo del SO.

Los hilos tienen aplicación en el desarrollo de programas que necesiten realizar varias tareas de forma simultánea, por ejemplo:

- Un procesador de textos puede comprobar la ortografía y la gramática del texto mientras que el usuario escribe, además de guardar el documento en disco cada cierto tiempo de forma automática.
- Un servidor web realiza su tarea de forma más eficiente si es capaz de atender varias peticiones entrantes de forma simultánea.

2 CREACIÓN Y EJECUCIÓN DE HILOS EN JAVA

En Java existen dos formas de crear hilos:

- Utilizando la clase java.lang.Thread que forma parte del API de concurrencia de bajo nivel que se suministra con la plataforma Java desde sus primeras versiones.
- Utilizando las APIs de concurrencia de alto nivel introducidas en la plataforma Java a partir de la versión 5.0.

Esta sección se ocupa únicamente de explicar la primera opción. La segunda se tratará al final de la unidad.

La clase Thread no es abstracta, no es final e implementa la interface Runnable. Esto unido a la amplia variedad de constructores que define, hace que dispongamos de múltiples posibilidades a la hora de crear hilos, y todas ellas derivadas de dos procedimientos básicos:

- Instanciar directamente la clase Thread.
- Instanciar una subclase de ésta.

La elección de uno u otro procedimiento determina la forma en la que se implementa la tarea que se ejecutará en el hilo y qué constructores de la clase Thread se podrán usar para construirlo.

De la misma forma que un sistema operativo multiproceso se encarga de planificar la ejecución concurrente de varios procesos, la JVM se encarga de planificar la ejecución concurrente de varios hilos dentro de un programa multihilo asignando recursos de CPU a cada uno de ellos en función de su estado. Un hilo puede encontrarse en uno de varios estados posibles. Cuando se crea, su estado inicial es "nuevo" y el planificador no le asignará recursos de CPU, es decir, no estará ejecutando la tarea asignada. Para ello es necesario que pase al estado "ejecutable", y eso sólo ocurrirá después de invocar a su método start, definido en la clase Thread. A partir de ese momento el hilo estará ejecutando su tarea en los instantes en los que el planificador le asigne recursos de CPU. El resto del tiempo se encontrará esperando a que eso ocurra o pasará a otros estados relacionados con la sincronización de hilos que se explicarán más adelante.

2.1 Crear hilos instanciando una subclase de Thread

La primera forma de crear hilos que vamos a estudiar consiste en instanciar clases que extiendan a la clase Thread. Los constructores de estas subclases invocarán implícita o explícitamente a uno de los constructores siguientes de la superclase:

```
Thread()
Thread(String name)
Thread(ThreadGroup group, String name)
```

Para implementar la tarea que se ejecutará en los hilos, las subclases redefinen el método run que heredan de Thread.

Ejemplo 1

Creamos una subclase de Thread que llamaremos UnHilo para crear hilos que muestren en la consola una secuencia de cinco mensajes numerados a intervalos de tiempo predefinidos. En cada mensaje se identificará el hilo que lo ha generado:

```
public class UnHilo extends Thread {

    public UnHilo(int id) {
        super("hilo " + id);
    }

    @Override
    public void run() {
        for (int i = 1; i <= 5; i++){
            try {
                Thread.sleep(100);
            } catch (InterruptedException e) {}
            System.out.println(getName() + ", mensaje " + i);
        }
    }
}
```

El constructor invoca a un constructor de Thread para asignar un nombre al hilo basado en el id que recibe como parámetro.

A continuación, se crea la clase Main para poner a prueba a la anterior creando y ejecutando tres hilos contadores:

```
public class Main {

    public static void main(String[] args) {
        for (int i = 1; i <= 3; i++)
            new UnHilo(i).start();
    }

}
```

La salida del programa nos puede dar una idea de cómo ha tenido lugar la ejecución concurrente de los hilos:

```
hilo 3, mensaje 1
hilo 1, mensaje 1
hilo 2, mensaje 1
hilo 3, mensaje 2
hilo 1, mensaje 2
hilo 2, mensaje 2
hilo 1, mensaje 3
hilo 2, mensaje 3
hilo 3, mensaje 3
hilo 1, mensaje 4
hilo 3, mensaje 4
hilo 2, mensaje 4
hilo 2, mensaje 5
hilo 3, mensaje 5
hilo 1, mensaje 5
```

Si ejecutamos varias veces el programa, veremos que el orden en que se intercalan los hilos para mostrar sus mensajes es diferente en cada ejecución. Esto es debido a que el planificador asigna tiempo de proceso a cada hilo mediante un proceso no determinista que depende de múltiples factores.

Cuando la tarea se implementa en unas pocas líneas de código y todos los hilos que la ejecutan se crean en la misma sentencia, puede ser interesante considerar la posibilidad de usar una clase anónima que extienda a la clase Thread.

Actividad 1

Reproduce el ejemplo 1 y añade comentarios que te ayuden a entenderlo. Después:

- Ejecuta el programa varias veces y comprueba que su salida varía de una ejecución a otra.
- Modifica el ejemplo para que los hilos se creen mediante una clase anónima.
- Partiendo de nuevo del ejemplo original, haz las modificaciones necesarias para que el programa lea de teclado los parámetros de funcionamiento (número de hilos, número de mensajes que mostrará cada hilo y tiempos de espera de cada hilo).
- Ejecuta el programa con diferentes valores de los parámetros de funcionamiento y expón tus conclusiones acerca de la concurrencia en este ejemplo basadas en la observación de las salidas en cada ejecución.

2.2 Crear hilos instanciando la clase Thread

Para crear un hilo instanciando directamente esta clase, tendremos que invocar a uno de los constructores siguientes:

```
Thread(Runnable target)
Thread(Runnable target, String name)
Thread(ThreadGroup group, Runnable target)
Thread(ThreadGroup group, Runnable target, String name)
Thread(ThreadGroup group, Runnable target, String name, long stackSize)
```

Obsérvese que todos ellos tienen en común el parámetro target de tipo Runnable, una interface que representa la tarea que ejecutará el hilo. Por tanto, esta forma de crear hilos implica la realización de los pasos siguientes:

- Crear una clase que implemente Runnable para definir una determinada tarea.
- Crear un objeto Runnable instanciando esta nueva clase.
- Instanciar la clase Thread pasándole al constructor la referencia al objeto creado en el punto anterior.

Los objetos de tipo Runnable no son hilos, sino que únicamente representan tareas que se van a ejecutar en un hilo. Sólo los objetos de tipo Thread son hilos.

Ejemplo 2

Creamos la clase Tarea que implementa Runnable para definir la misma funcionalidad que ejecutaban los hilos en el ejemplo 1:

```
public class Tarea implements Runnable {
    public void run() {
        for (int i = 1; i <= 5; i++){
            try {
                Thread.sleep(100);
            } catch (InterruptedException e) {
            }
            System.out.printf("%s, mensaje %d\n", Thread.currentThread().getName(), i);
        }
    }
}
```

A continuación, creamos la clase Main en la que finalmente se crearán hilos que ejecutarán la tarea definida en la clase Tarea:

```
public class Main {
   public static void main(String[] args){
      Runnable tarea = new Tarea();
      for (int i = 1; i <= 3; i++)
         new Thread(tarea, "hilo " + i).start();
   }
}
```

En el método main se pueden ver los dos últimos pasos para completar la creación de cada hilo:

- Primero se crea un objeto Runnable instanciando la clase Tarea.
- Finalmente se crea el hilo instanciando la clase Thread. Al constructor se le pasa la referencia al objeto Runnable y el nombre que tendrá el hilo.

Actividad 2

Reproduce el ejemplo anterior y añade comentarios que te ayuden a entenderlo. Después:

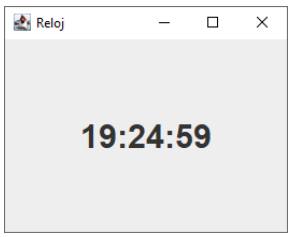
- Ejecuta el ejemplo y compara los resultados con los obtenidos en el ejemplo anterior. ¿Existe alguna diferencia? Razona la respuesta.
- Haz las modificaciones necesarias para que el programa lea de teclado los parámetros de funcionamiento (número de hilos, número de mensajes que mostrará cada hilo y tiempos de espera de cada hilo).

El uso de la interface Runnable es una de las alternativas que resultan adecuadas cuando, por la razón que sea, decidimos implementar la funcionalidad de los hilos en una clase que extiende a otra clase distinta de Thread.

Ejemplo 3

Crear una aplicación que muestre un reloj en pantalla en formato digital según las especificaciones siguientes:

- Interfaz gráfica de usuario basada en Swing:



- Desarrollo de la aplicación en una única clase que extienda a la clase JFrame.
- Actualización de la hora cada segundo mediante un proceso repetitivo que se ejecutará en un hilo. Este hilo se creará e iniciará después de que la ventana de la aplicación se haga visible.

Teniendo en cuenta estas especificaciones, vamos a crear una clase llamada Reloj que extiende a JFrame e implementa Runnable para definir la funcionalidad del hilo que actualizará la hora:

```
public class Reloj extends JFrame implements Runnable {

    private static final long serialVersionUID = 1L;
    private DateTimeFormatter formatter = DateTimeFormatter.ofPattern("HH:mm:ss");
    private JLabel hora;

    public Reloj() {
        super("Reloj");
        hora = new JLabel(formatter.format(LocalDateTime.now()));
        hora.setBorder(BorderFactory.createCompoundBorder(
                BorderFactory.createEmptyBorder(70, 70, 70, 70),
                hora.getBorder()
                ));
        hora.setFont(hora.getFont().deriveFont(30f));
        hora.setHorizontalAlignment(JLabel.CENTER);
        getContentPane().add(hora);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        pack();
        setLocationRelativeTo(null);
    }

    public void run() {
        Runnable actualizarHora = new Runnable() {
            public void run() {
                hora.setText(formatter.format(LocalDateTime.now()));
            }
        };
        while (true) {
            SwingUtilities.invokeLater(actualizarHora);
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {}
        }
    }

    private void iniciar() {
        setVisible(true);
        new Thread(this, "segundero").start();
    }

    public static void main(String[] args) {
        SwingUtilities.invokeLater(new Runnable() {
            public void run() {
                new Reloj().iniciar();
            }
        });
    }

}
```

En el método main se instancia la propia clase Reloj y se invoca al método de instancia iniciar. En él se hace visible la ventana de la aplicación y se crea el hilo a partir de la referencia al objeto Reloj, que obviamente también es un Runnable.

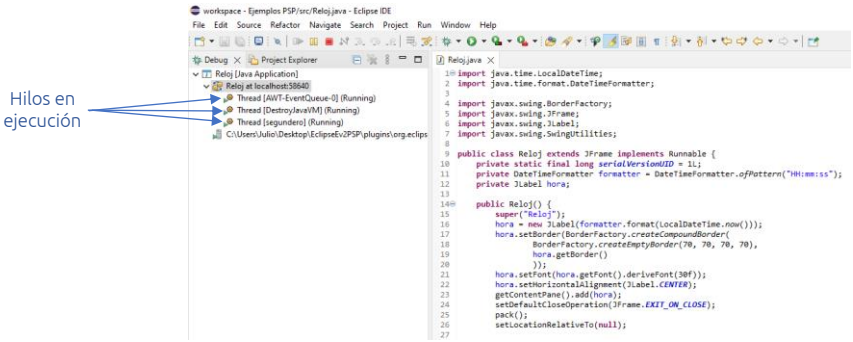
La interface Runnable puede tener utilidad más allá de la creación de nuevos hilos. Este hecho se pone de manifiesto en este ejemplo en aquellas líneas donde se usa la clase SwingUtilities.

Como no podía ser de otra forma, también existe la posibilidad de implementar Runnable en una clase anónima y, habida cuenta de que es una interface funcional, existen aún dos formas más de crear hilos:

- Usando expresiones lambda.
- Usando referencias a métodos: en una clase se puede definir múltiples tareas, cada una en un método de clase o en un método de instancia sin importar cómo se llamen, siempre que retornen void y no declaren parámetros formales. Para crear los hilos, en el constructor de la clase Thread se especifica el objeto Runnable mediante una referencia al método que implemente la tarea correspondiente.

Actividad 3

- Reproduce el ejemplo 3, añade los comentarios que te ayuden a entenderlo y ejecútalo en modo depuración desde el IDE que utilices. En la vista de depuración observa los hilos que están en ejecución:



- Averigua para qué sirve el método SwingUtilities.invokeLater(…) ¿Cómo demostrarías que las referencias a los objetos Runnable que se le pasan como parámetro no se usan para crear nuevos hilos?
- Modifica este ejemplo para pasar la funcionalidad del hilo que actualiza la hora de las tres formas que se proponen a continuación:
  - Con una expresión lambda.
  - Con una referencia a un método.

2.3 Esperar a que otro hilo finalice su ejecución

El método join de la clase Thread se usa para que un hilo espere a que otro hilo finalice su ejecución. Para usar este método, un hilo (el que va a esperar) usa una referencia a otro hilo (el que tiene que finalizar) para invocarlo.

Ejemplo 4

En este ejemplo el método main crea un hilo y después espera a que el hilo que ha creado finalice:

```
public class EjemploJoin {
    public static void main(String[] args) throws InterruptedException {
        Thread hilo = new Thread("cuenta atrás") {
            @Override
            public void run() {
                for (int i = 5; i >= 0; i--) {
```

```
                    System.out.println(i);
                    try {
                        Thread.sleep(1000);
                    } catch (InterruptedException e) {
                    }
                }
                System.out.println("hilo finalizado");
            }
        };
        hilo.start();
        hilo.join();
        System.out.println("método main finalizado");
    }
}
```

Para observar el funcionamiento de join se ejecuta el programa dos veces. Antes de ejecutarlo la segunda vez, se comenta la línea donde se invoca a join. Se puede ver, comparando las salidas, que en el primer caso el método main no finaliza hasta que lo hace el hilo "cuenta atrás" al contrario que en el segundo caso:

Con join

```
5
4
3
2
1
0
hilo finalizado
método main finalizado
```

Sin join

```
método main finalizado
5
4
3
2
1
0
hilo finalizado
```

Actividad 4

Reproduce el ejemplo 4, añade los comentarios que te ayuden a entenderlo y ejecútalo una vez usando el método join y otra más sin usarlo. Comprueba, usando el modo depuración, que en el segundo caso aparece el ya conocido hilo "DestroyJavaVM" cuando finaliza el método main y antes de que finalice el hilo "cuenta atrás". Comprueba también que esto no ocurre en el primer caso.

Actividad 5

Crea un programa que reciba a través de la línea de comandos los parámetros siguientes:

- Una ruta a una carpeta que contenga varios ficheros de texto.
- A continuación, un indicador que admitirá únicamente dos valores: S o C.

El objetivo del programa es contar cuántas líneas, palabras y caracteres contiene cada fichero de texto. El indicador determinará si esto se hará procesando los ficheros de forma secuencial (indicador S) en el hilo principal, o concurrentemente usando un hilo para cada fichero (indicador C).

En cualquier caso, el programa medirá el tiempo total de proceso de todos los ficheros y finalizará mostrando todos los datos obtenidos en la salida estándar.

Haz una prueba de procesamiento secuencial y otra de procesamiento concurrente usando los mismos ficheros de texto en ambos casos. Una vez que tengas los resultados, comparte con el resto de la clase tus datos y conclusiones acerca del rendimiento obtenido en ambos casos.

Notas:

- Sólo se considerará como palabra aquella secuencia de caracteres que encaje en la expresión regular \p{L}+.
- En https://www.gutenberg.org/ puedes encontrar y descargar libros gratuitos en formato de texto plano de gran tamaño como, por ejemplo, "El Quijote".

3 GESTIÓN DE HILOS

Dentro de los diferentes aspectos relacionados con la gestión de hilos, en esta sección vamos a tratar los siguientes:

- Gestión de las prioridades.
- Control de su ejecución: interrupción, suspensión y reanudación.

La clase Thread define métodos para suspender, reanudar y detener la ejecución de un hilo. Sin embargo, han sido declarados obsoletos en el API de Java por ser propensos a provocar interbloqueos y no se recomienda su uso. Por tanto, en esta sección se desarrollarán métodos seguros para gestionar la suspensión, reanudación e interrupción de la ejecución de los hilos.

3.1 Otros métodos de la clase Thread

A continuación, se muestra un resumen de los métodos de la clase Thread que permiten manipular los hilos una vez creados:

`void start()`
Hace que el planificador inicie la ejecución del hilo mediante una llamada *callback* al método `run`.
`boolean isAlive()`
Retorna `true` si el hilo está vivo.
`boolean isDaemon()`
Retorna `true` si el hilo es un demonio.
`boolean isInterrupted()`
Comprueba si este hilo ha sido requerido para que interrumpa su ejecución.
`long getId()`
Devuelve el identificador de este hilo.
`String getName()`
Devuelve el nombre de este hilo.
`int getPriority()`
Retorna la prioridad de este hilo.
`void setName(String name)`
Cambia el nombre de este hilo, asignándole el especificado como argumento.
`void setPriority(int p)`
Establece la prioridad del hilo al valor p.
`void setDaemon(boolean on)`
Convierte al hilo en un demonio (`on=true`) o en un hilo de usuario (`on=false`).
`void interrupt()`
Traslada a este hilo la petición de que interrumpa su ejecución.
`static boolean interrupted()`
Comprueba si el hilo actual ha recibido una petición para que interrumpa su ejecución.

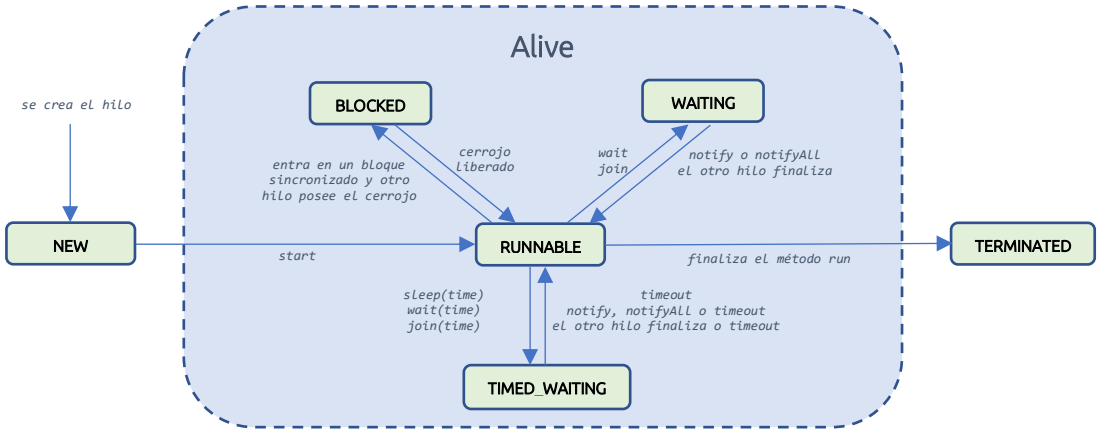
`boolean isInterrupted()`
Comprueba si este hilo ha recibido una petición para que interrumpa su ejecución.
`static void yield()`
Hace que el hilo actual de ejecución pare temporalmente y permita que otros hilos se ejecuten.
`static Thread currentThread()`
Devuelve una referencia al hilo actual.
`static void sleep(long milis) / static void sleep(long milis, int nanos)`
`Duerme al hilo actual durante el número de milisegundos especificado.`
`static void holdsLock(Object obj)`
`Retorna true si el hilo actual posee el cerrojo del monitor sobre el objeto especificado.`

3.2 Estados de un hilo

Durante su ciclo de vida, un hilo puede pasar por varios estados. En la tabla siguiente se enumeran todos los estados posibles, cada uno representado por un valor simbólico definido en la enumeración Thread.State:

`Thread.State.NEW`	El hilo ha sido creado, pero aún no ha invocado al método start.
`Thread.State.RUNNABLE`	El hilo está en ejecución después de haber invocado al método start.
`Thread.State.BLOCKED`	El hilo está bloqueado a la espera de adquirir el bloqueo de un monitor.
`Thread.State.WAITING`	El hilo se encuentra esperando indefinidamente debido a que ha invocado al método join o al método wait.
`Thread.State.TIMED_WAITING`	El hilo se encuentra esperando un tiempo máximo debido a que ha invocado a uno de los métodos siguientes: sleep(long milis) sleep(long milis, int nanos) wait(long milis) wait(long milis, int nanos) join(long milis) join(long milis, int nanos)
`Thread.State.TERMINATED`	El hilo ha completado su ejecución.

En el diagrama siguiente se representa el ciclo de vida de los hilos mostrando todos los estados posibles y las transiciones permitidas entre unos y otros:



Cuando se crea un hilo su estado inicial es NEW y el planificador sólo le asignará tiempo de CPU cuando pase a estado RUNNABLE. Eso ocurrirá cuando se invoque a su método start. La transición del estado RUNNABLE hacia el estado TERMINATED se produce cuando el hilo completa su tarea y el método run finaliza. El resto de las transiciones se tratarán cuando se estudien los mecanismos básicos de sincronización de hilos.

3.3 Gestión de prioridades

En un programa Java, cada hilo tiene una prioridad. Por defecto, un hilo hereda la prioridad del hilo padre que lo crea y ésta se puede modificar y consultar invocando a los métodos setPriority y getPriority de la clase Thread respectivamente.

La prioridad se representa mediante un número entero entre 1 y 10, siendo el valor 1 la mínima prioridad y 10 la máxima. Las constantes enteras Thread.MIN_PRIORITY, Thread.MAX_PRIORITY y Thread.NORM_PRIORITY representan la prioridad mínima (1), máxima (10) y normal (5) respectivamente.

Si una misma CPU tiene que ejecutar varios hilos, el planificador elige qué hilo debe ejecutarse en función de la prioridad asignada. En general, se ejecutarán antes los hilos de mayor prioridad que los de menor prioridad, y seguirán ejecutándose hasta que:

- Un hilo de mayor prioridad de convierta en ejecutable.
- Cedan voluntariamente el control a otros hilos de igual prioridad invocando al método yield de la clase Thread.
- Dejen de ser ejecutables, bien por haber concluido su ejecución, por entrar en estado de bloqueo o por entrar en estado de espera.

Cuando un hilo se convierte en ejecutable y no cede voluntariamente el control para que puedan ejecutarse otros hilos, se dice que es un "hilo egoísta". Algunos sistemas operativos combaten estas situaciones con una estrategia de planificación del tipo *time-slicing* en la que el tiempo de proceso se divide en porciones que se asignan alternativamente a los hilos ejecutables para crear la ilusión de concurrencia.

A la hora de crear programas multihilo con diferentes prioridades hemos de tener en cuenta que el comportamiento descrito no está garantizado y dependerá de la plataforma en la que se ejecuten los programas. En la práctica casi nunca hay que establecer a mano las prioridades.

Actividad 6

Modifica de nuevo el ejemplo 2 para que también se pueda especificar la prioridad de cada hilo a través de la entrada estándar.

Pruébalo con diferentes datos de entrada, procurando que haya hilos que se ejecuten con diferentes prioridades. Analiza los resultados y comenta cómo han influido las prioridades asignadas en la ejecución concurrente de los hilos.

3.4 Interrupción segura

Un hilo finaliza su ejecución de forma natural cuando a la conclusión de su tarea se produce el retorno del método run, pasando a estado TERMINATED. Para interrumpir el hilo antes de que concluya la tarea asignada, es necesario implementar algún mecanismo que permita que el retorno de este método se lleve a cabo de forma segura y ordenada.

Ya hemos visto que se desaconseja invocar al método stop del hilo para llevar a cabo esta tarea, no sólo por los problemas de interbloqueo que puede ocasionar, sino también porque constituye una forma brusca de detener la ejecución del hilo, por lo que será necesario implementar algún mecanismo de interrupción. Entre los muchos mecanismos de interrupción que se pueden implementar, los dos más utilizados son:

- Interrupción basada en un *flag* definido mediante una variable booleana.
- Interrupción basada en el método interrupt de la clase Thread.

En este apartado vamos a ver como implementar estos mecanismos cuando el hilo lleva a cabo su tarea mediante la ejecución de un bucle que itera indefinidamente.

3.4.1 Interrupción usando un *flag*

Consiste en usar una variable compartida de tipo booleano que indique si el hilo debe finalizar su ejecución.

Ejemplo 5

```
public class EjemploInterrumpirConFlag extends Thread {
    private volatile boolean finalizar = false;

    @Override
    public void run() {
        while (!finalizar) {
            System.out.println("en el Hilo");
            try {
                Thread.sleep(200);
            } catch (InterruptedException e) {
            }
        }
        System.out.println("hilo finalizado");
    }

    public void finalizar() {
        finalizar = true;
    }

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

La razón por la que la variable finalizar se ha declarado volatile está relacionada con el modelo de memoria de Java. El ejemplo siguiente, muestra lo que puede ocurrir si no se hace así:

```java
public class EjemploInterrumpirConFlagNoVolatile extends Thread {
    static boolean finalizar;

    @Override
    public void run() {
        while (!finalizar);
        System.out.println("hilo finalizado");
    }

    public static void main(String[] args) throws InterruptedException {
        Thread t = new EjemploInterrumpirConFlagNoVolatile();
        t.start();
        Thread.sleep(3000);
        finalizar = true;
        System.out.println("main finalizado");
    }
}
```

Si ejecutamos este programa, veremos que después de 3 segundos solo se muestra el mensaje "main finalizado" y el programa no finaliza como cabría esperar.

Actividad 7

Prueba las dos variantes del ejemplo 5 y comprueba que efectivamente, al ejecutar la segunda se manifiesta el problema descrito.

Actividad 8

Visita los enlaces siguientes para aprender más acerca del modificador volatile:

- Java Volatile Keyword
- Guide to the Volatile Keyword in Java

Haz un resumen de lo que has aprendido acerca del modelo de memoria de Java que incluya una explicación de por qué el segundo ejemplo no funciona como se esperaba y por qué se soluciona con el uso de volatile.

Como alternativa al uso de volatile, está la posibilidad de utilizar las clases del API de concurrencia de alto nivel que representan variables atómicas. Por ejemplo, en lugar de utilizar como *flag* una variable booleana, se usaría un objeto de la clase AtomicBoolean.

3.4.2 Interrupción usando el método interrupt

Cuando se invoca al método interrupt de un hilo en ejecución, ésta no se interrumpe realmente. En su lugar se activa un indicador interno que le insta a que lo haga voluntariamente. Se dice que un hilo es obediente si acata la petición. Para ello tendrá que comprobar de forma periódica si el indicador de interrupción está activado invocando al método isInterrupted o al método interrupted. Después de invocar a cualquiera de estos dos métodos el indicador de interrupción se desactiva automáticamente.

Los métodos sleep y join de la clase Thread y los métodos wait de la clase Object declaran el lanzamiento de la excepción InterruptedException. Esta excepción es de tipo *checked* y, por tanto, hay que capturarla o a declarar su lanzamiento.

Estos métodos lanzan la excepción si:

- Un hilo invoca al método interrupt de otro hilo que encuentra en estado WAITING o TIMED_WAITING como consecuencia de haber invocado a alguno de estos métodos. El segundo retornará inmediatamente al estado RUNNABLE para el tratamiento de la excepción y, eventualmente, acatar la petición de interrupción.
- Se invoca al método interrupt de un hilo en estado RUNNABLE activando su indicador interno de interrupción. Después, ese mismo hilo invoca a alguno de estos métodos antes

de hacer algo que pueda desactivar el indicador interno de interrupción, provocando que inmediatamente lancen la excepción. En consecuencia, el hilo no pasará a estado WAITING o TIMED_WAITING, sino que se mantendrá estado RUNNABLE para el tratamiento de la excepción y, eventualmente, acatar la petición de interrupción.

Esta excepción no se lanza para poner de manifiesto la ocurrencia de una condición de error. En su lugar se utiliza como mecanismo para que un hilo que se encuentra en estado de espera tenga la oportunidad de ser obediente inmediatamente, es decir, sin que se complete la espera programada.

Se ha de tener en cuenta que el lanzamiento de esta excepción desactiva automáticamente el indicador interno de interrupción. En consecuencia, un hilo obediente podría tener que restaurarlo cuando capture la excepción invocando el mismo a su método interrupt. A continuación, veremos algunos ejemplos.

Ahora que conocemos todos los detalles del funcionamiento del método interrupt, vamos a ver cómo usarlo para interrumpir de forma segura la ejecución de un hilo.

Ejemplo 6

```
public class EjemploInterrupt extends Thread {
    public void run() {
        while (!isInterrupted())
            System.out.println("en el hilo");
    }

    public static void main(String[] args) throws InterruptedException {
        EjemploInterrupt h = new EjemploInterrupt();
        h.start();
        Thread.sleep(2000);
        h.interrupt();
        h.join();
        System.out.println("hilo finalizado");
    }
}
```

A continuación, se muestra una modificación del ejemplo anterior en la que el hilo duerme durante dos décimas de segundo cada vez que muestra el mensaje.

Ejemplo 7

```
public class EjemploInterrupt extends Thread {
    public void run() {
        while (!isInterrupted()) {
            System.out.println("en el Hilo");
            try {
                Thread.sleep(200);
            } catch (InterruptedException e) {
                System.out.println("interrumpido mientras dormía");
                interrupt();
            }
        }
    }

    public static void main(String[] args) throws InterruptedException {
        EjemploInterrupt h = new EjemploInterrupt();
        h.start();
        Thread.sleep(2000);
        h.interrupt();
        h.join();
        System.out.println("hilo finalizado");
    }
}
```

También se podría pensar en ejecutar una sentencia break dentro del bloque catch como alternativa a la reactivación del indicador interno de interrupción. Esta práctica sólo será válida si la ejecución parcial de la iteración en curso es aceptable para un finalización ordenada y segura.

Actividad 9

Reproduce el ejemplo 7, añade los comentarios que te ayuden a entenderlo y después modifícalo según las especificaciones siguientes:

- El hilo incrementará un contador cada 200 milisegundos en lugar de mostrar el mensaje.
- Después de crear el hilo, main esperará a que el usuario teclee la palabra "fin", después finalizará el hilo secundario y, cuando lo haya hecho, mostrará el valor del contador.

Responde a las preguntas siguientes:

- ¿Se mostrará siempre el mensaje "interrumpido mientras dormía" cuando finaliza el hilo?
- ¿Sería aceptable el uso de break en lugar de interrupt cuando se captura la excepción InterruptedException?

Razona todas las respuestas.

3.5 Suspensión y reanudación seguras

Para suspender temporalmente la ejecución de un hilo y reanudarla de forma segura, es necesario usar mecanismos de sincronización de hilos que se explicarán con más detalle en posteriores apartados. La clase siguiente muestra un ejemplo de cómo implementar un mecanismo de suspensión y reanudación de la ejecución de un hilo.

Ejemplo 8

```
public class EjemploSuspension extends Thread {
    static boolean suspendido = false;

    public synchronized void suspender() {
        suspendido = true;
    }

    public synchronized void reanudar() {
        suspendido = false;
        notify();
    }

    @Override
    public void run() {
        while (true) {
            try {
                synchronized (this) {
                    if (suspendido) {
                        System.out.println("suspendido");
                        wait();
                    }
                }
                System.out.println("en ejecución");
                Thread.sleep(1000);
            } catch (InterruptedException e) {
            }
        }
    }

    public static void main(String[] args) throws IOException, InterruptedException {
        BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
        EjemploSuspension hilo = new EjemploSuspension();
        hilo.start();
        do {
            String opcion = in.readLine();
            System.out.println("s -> suspender / r -> reanudar");
            switch (opcion.toLowerCase()) {
            case "s":
                hilo.suspender();
                break;
            case "r":
                hilo.reanudar();
                break;
            }
        } while(true);
    }
}
```

Actividad 10

Prueba el ejemplo 8 y verifica su funcionamiento. Después, haz las modificaciones necesarias para añadir una opción más que permita interrumpir el hilo y finalizar el programa de forma segura y ordenada.

Investiga acerca de por qué no se ha declarado la variable suspendido como volatile.

4 SINCRONIZACIÓN

La sincronización de hilos consiste en la implementación de mecanismos que permitan solucionar problemas de diferente naturaleza derivados del acceso concurrente a recursos compartidos.

Cuando varios hilos se ejecutan concurrentemente y compiten por acceder a un recurso compartido se pueden producir inconsistencias en los datos. Esta situación se conoce como **condición de carrera** o **race condition**. Un conjunto de sentencias cuya ejecución concurrente puede dar lugar a condiciones de carrera recibe el nombre de **sección crítica**. Las condiciones de carrera representan un problema clásico de la programación concurrente que se resuelve mediante la implementación de mecanismos de sincronización que tienen como objetivo anular la concurrencia en las secciones críticas.

Las condiciones de carrera no son el único problema derivado del acceso concurrente a recursos compartidos. También pueden existir problemas de interdependencia entre hilos que se resuelven mediante la implementación de mecanismos adicionales de sincronización.

Java permite una sincronización de hilos basada en la implementación de tres mecanismos básicos:

- Exclusión mutua.
- Parada y espera.
- Comunicación entre hilos.

Todos ellos precisan de la delimitación de bloques de código sincronizados ligados a la existencia de secciones críticas.

4.1 Exclusión mutua.

La exclusión mutua es un mecanismo que permite anular la concurrencia dentro de una sección crítica. Antes de tratarlo más en detalle, vamos a ver un ejemplo que ilustra cómo una condición de carrera genera un problema de consistencia.

Ejemplo 9

Programa en el que varios hilos incrementan un contador concurrentemente:

```
public class Contador {
    private int n;

    public Contador(int n) {
        this.n = n;
    }

    public void inc() {
        n++;
    }

    public int get() {
        return n;
    }
}
```

```
public class Main {
    private static Contador c = new Contador(100);
```

```
    private static void run() {
        for (int i = 0; i < 100; i++) {
            c.inc();
            try {
                Thread.sleep(10);
            } catch (InterruptedException e) {}
        }
    }

    public static void main(String[] args) throws InterruptedException {
        Thread t1 = new Thread(Main::run);
        Thread t2 = new Thread(Main::run);
        t1.start();
        t2.start();
        t1.join();
        t2.join();
        System.out.println("Contador = " + c.get());
    }
}
```

La salida de este programa debería ser "Contador = 300". Sin embargo, ejecutándolo varias veces veremos en la mayoría de las ocasiones su salida será un valor inferior a 300. Esto se debe a que la sentencia n++ en el método inc de la clase Contador no se ejecuta como una operación atómica. Traducida a lenguaje máquina, esta sentencia se podría descomponer en tres instrucciones:

- Lectura de la memoria para transferir el valor de n a un registro de la CPU.
- Incremento en una unidad del valor almacenado en el registro de la CPU.
- Escritura en memoria del valor almacenado en el registro de la CPU para actualizar el valor de n.

Según esto, un hilo puede ser interrumpido para ceder el control de la CPU a otro hilo habiendo ejecutado parcialmente la sentencia n++. Por ejemplo, después de haber ejecutado sólo la primera o las dos primeras instrucciones máquina. Si el hilo que adquiere el control de la CPU ejecuta la misma sentencia, leerá el mismo valor de n que su predecesor y en consecuencia, ambos hilos habrán incrementando el mismo valor del contador y habrán escrito en la memoria el mismo resultado. Lo que deberían haber sido dos incrementos del contador, se ha quedado en un único incremento.

La solución a este problema se basa en uso del mecanismo de exclusión mutua en uno o varios bloques de código sincronizados. Cada bloque de código sincronizado delimita el comienzo y el fin de una sección crítica y dentro de él, la exclusión mutua se consigue de la forma siguiente:

- El acceso al bloque sincronizado se controla mediante un **cerrojo** que cualquier hilo puede adquirir para bloquear el acceso de otros hilos a dicho bloque.
- El cerrojo estará gestionado por un objeto que actuará como **monitor** del bloque sincronizado.
- Cualquier objeto puede ser designado para actuar como monitor del bloque sincronizado.
- Los monitores también implementan los mecanismos básicos de parada y espera, y de comunicación entre hilos que sólo estarán disponibles dentro de los bloques sincronizados que monitorizan.
- Cuando el flujo de ejecución de un hilo *A* llega a un bloque sincronizado, ocurrirá una de las dos situaciones siguientes:
    - Si ningún hilo posee el cerrojo, el hilo *A* lo obtiene del monitor y comienza a ejecutar el bloque sincronizado.
    - Si otro hilo *B* posee el cerrojo, el hilo *A* pasa a estado bloqueado, permaneciendo en ese estado hasta que *B* libere el cerrojo. En ese momento *A* volverá a competir por la adquisición del cerrojo.
- El cerrojo se libera cuando:

    - El hilo que lo posee completa la ejecución del bloque sincronizado.
    - El hilo que lo posee pasa voluntariamente a uno de los estados `Thread.State.WAITING` o `Thread.State.TIMED_WAITING` usando mecanismos proporcionados por el monitor.
    - Dentro del bloque sincronizado se lanza una excepción que no se captura.
- La adquisición de un cerrojo bloquea el acceso concurrente a todos los bloques sincronizados monitorizados por un mismo monitor.

Para definir un bloque sincronizado y establecer cuál va a ser su monitor se usará, como era previsible, la palabra reservada synchronized. Existen dos posibilidades:

- Sincronización a nivel de método. En este caso synchronized se usa como modificador en la cabecera de un método y el bloque sincronizado abarca todo el cuerpo del método. La designación del monitor se realiza de forma implícita y dependerá del tipo de método:
    - Método de clase: el monitor es el objeto referenciado por el atributo class de la clase.
    - Método de instancia: el monitor es el objeto con el que se invoca al método.
- Sincronización a nivel de bloque. Se usa la sintaxis siguiente:

```
synchronized (monitor) {
    sentencias de la sección crítica …
}
```

  La designación del monitor se realiza de forma explícita con el parámetro monitor. Este parámetro ha de ser una referencia estable a un objeto elegido por el programador, es decir, una referencia que una vez definida no cambie.

Dado que con la sincronización se pierde la ventaja de la concurrencia, es recomendable que el tamaño de los bloques sincronizados no se extienda más allá de lo estrictamente necesario.

Ejemplo 10

Es una modificación del ejemplo 9 que representa una solución correcta del problema del contador usando el mecanismo de exclusión mutua implementado en Java:

```
public class Contador {
    private int n;

    public Contador(int n) {
        this.n = n;
    }

    public synchronized void inc() {
        n++;
    }

    public int get() {
        return n;
    }
}
```

```
public class Main {
    private static Contador c = new Contador(100);

    private static void run() {
        for (int i = 0; i < 100; i++) {
            c.inc();
            try {
                Thread.sleep(10);
            } catch (InterruptedException e) {
            }
        }
    }
```

```
    public static void main(String[] args) throws InterruptedException {
        Thread t1 = new Thread(Main::run);
        Thread t2 = new Thread(Main::run);
        t1.start();
        t2.start();
        t1.join();
        t2.join();
        System.out.println("Contador = " + c.get());
    }
}
```

En este caso el monitor será el objeto referenciado por atributo c de la clase Main.

Aunque conceptualmente el mecanismo de exclusión mutua pueda parecer relativamente simple, su aplicación práctica a la resolución de problemas de concurrencia puede resultar compleja. De hecho, su uso incorrecto puede derivar en cierto tipo de errores difíciles de detectar y de depurar, entre los que se encuentran los que se describen a continuación:

- Interbloqueo o *deadlock*: Sean dos hilos *H1* y *H2* y dos recursos compartidos *R1 y R2* con acceso sincronizado. Supongamos que H1 obtiene el cerrojo para acceder a *R1* y *H2* adquiere el cerrojo para acceder a *R2*. Supongamos ahora que después de que ambos hayan adquirido ambos cerrojos y antes de que finalicen la ejecución del bloque sincronizado, *H1* necesita acceder a *R2* y *H2* necesita acceder a *R1*. En esta situación ninguno podrá avanzar debido a que cada uno se bloqueará hasta que el otro libere su cerrojo, dando lugar una situación de interbloqueo de la que ya no podrán salir.
- Bloqueo activo o *livelock*: es similar al interbloqueo, pero los estados de los hilos involucrados cambian constantemente de tal forma que ninguno permanece bloqueado indefinidamente, pero ninguno progresa.
- Inanición o *starvation*: Sean tres hilos, *H1, H2* y *H3*, accediendo periódicamente a un recurso compartido. Supongamos que *H1* posee el cerrojo del recurso y que *H2* y *H3* están bloqueados a la espera del cerrojo. Cuando *H1* haya ejecutado su sección crítica tanto *H2* como *H3* podrán solicitar el cerrojo. Supongamos ahora que cada vez que se libera el cerrojo siempre son *H1* y *H2* los que lo obtienen. Ocurrirá entonces que *H3* nunca accede al recurso produciéndose una situación de inanición para él.

Actividad 11

Reproduce los dos ejemplos 9 y 10, verificando que en el primero se manifiesta el problema de inconsistencia descrito y que éste se ha solucionado en el segundo.

A parte de la solución basada en exclusión mutua, ¿se te ocurre alguna otra solución?

Actividad 12

Busca en Internet ejemplos simples de Interbloqueo, bloqueo activo e inanición que puedas reproducir en Java.

4.2 Parada y espera. Comunicación entre hilos

En esta sección vamos a tratar dos mecanismos básicos de sincronización que resultarán útiles para resolver problemas de concurrencia en los que se pone de manifiesto alguna forma de interdependencia entre hilos que haga que el avance de unos hilos dependa de que otros hayan completado una determinada tarea, normalmente relacionada con el acceso a recursos compartidos.

Un ejemplo clásico de interdependencia entre hilos es el problema de los productores y los consumidores. En este problema varios hilos acceden concurrentemente a un recurso compartido, un almacén de productos con una capacidad máxima. Unos hilos actúan como productores

depositando productos en el almacén, mientras que otros actúan como consumidores retirando productos del almacén. En todo este proceso se ha de tener en cuenta que:

- El almacén es un recurso compartido y será necesario emplear el mecanismo de exclusión mutua para acceder a él.
- Un consumidor no puede avanzar si se encuentra el almacén vacío.
- Un productor no puede avanzar si se encuentra el almacén lleno.

Por tanto, para solucionar el problema es necesario utilizar tanto un mecanismo de parada y espera, como un mecanismo de comunicación entre hilos. La utilización conjunta de estos dos mecanismos básicos de sincronización permite implementar un mecanismo de parada y avance que se asegurará de que:

- Cuando un productor vaya a almacenar un producto y el almacén esté lleno, libere el cerrojo y espere hasta que al menos un consumidor retire un producto.
- Cuando un consumidor vaya a retirar un producto y el almacén esté vacío, libere el cerrojo y espere hasta que al menos un productor almacene un producto.
- Cuando un productor almacene un producto, se lo notifique a los consumidores que estén en estado de espera para que retornen al estado ejecutable y vuelvan a competir por el acceso al almacén.
- Cuando un consumidor retire un producto, se lo notifique a los productores que estén en estado de espera para que retornen al estado ejecutable y vuelvan a competir por el acceso al almacén.

Java implementa en la clase Object estos dos mecanismos básicos de sincronización. El de parada y espera en los métodos wait y el de comunicación entre hilos en los métodos notify y notifyAll. Se usan de la forma siguiente:

- Si un hilo *A* se encuentra ejecutando un bloque sincronizado y tiene esperar a que otro hilo *B* complete una tarea, el hilo *A* usará el monitor para invocar al método wait. Entonces *A* pasará al estado WAITING, liberando el cerrojo para que otros hilos puedan ejecutar el bloque sincronizado.
- Cuando el hilo *B* complete su tarea, notificará otros hilos que puede continuar. Esto lo hará invocando al método notify o notifyAllcon el mismo monitor con el que *A* invoco al método wait. A partir de ese momento, *A* recuperará el estado RUNNABLE.
- La ejecución de cualquiera de estos métodos fuera de un bloque sincronizado o con el monitor equivocado, provocará el lanzamiento de la excepción IllegalMonitorStateException.

Para ponerlos en práctica, vamos a resolver una versión simplificada del problema de los productores y consumidores según las especificaciones siguientes:

- Existirán un único productor y consumidor.
- Tanto productor como consumidor llevarán a cabo su tarea cada cierto número de milisegundos.
- Ambos hilos permanecerán realizando su tarea por tiempo indefinido.

Ejemplo 11

Comenzamos creando las clases Productor y Consumidor:

```
public class Productor extends Thread {
    private long retardo, contador = 0;
    private Almacen almacen;

    public Productor(Almacen almacen, long retardo) {
```

```
        super(String.valueOf(id));
        this.retardo = retardo;
        this.almacen = almacen;
    }

    public void run(){
        while (true) {
            String producto = String.format("%d", ++contador);
            almacen.almacenar(producto);
            System.out.println("producto " + producto + " almacenado");
            try { Thread.sleep(retardo); } catch (InterruptedException e) {}
        }
    }
}
```

```
public class Consumidor extends Thread {
    private long retardo;
    private Almacen almacen;

    public Consumidor(Almacen almacen, long retardo) {
        this.retardo = retardo;
        this.almacen = almacen;
    }

    public void run(){
        while (true) {
            String producto = almacen.retirar();
            System.out.println("producto " + producto + " retirado");
            try { Thread.sleep(retardo); } catch (InterruptedException e) {}
        }
    }
}
```

El constructor de productor y consumidor reciben el tiempo que deberán esperar cada vez que vayan a acceder al almacén.

Después creamos la clase Almacen, que representa el recurso compartido, donde se usarán los mecanismos de sincronización:

```
public class Almacen {

    private int almacenados = 0;
    private String [] productos;

    public Almacen(int capacidad) {
        productos = new String[capacidad];
    }

    public synchronized void almacenar(String producto) {
        if (almacenados == productos.length) // almacén lleno
            try {
                wait();
            } catch (InterruptedException e) {}
        productos[almacenados++] = producto;
        notify();
    }

    public synchronized String retirar() {
        if (almacenados == 0) // almacén vacío
            try {
                wait();
            } catch (InterruptedException e) {}
        String producto = productos[--almacenados];
        notify();
        return producto;
    }
```

```
}
```

Finalmente se crea la clase Main donde se crea el almacén, el hilo productor, el hilo consumidor y se ejecutan estos hilos para iniciar la simulación:

```
public class Main {
    public static void main(String[] args) {
        Almacen almacen = new Almacen(10);
        Productor productor = new Productor(almacen, 100);
        Consumidor consumidor = new Consumidor(almacen, 1000);
        productor.start();
        consumidor.start();
    }
}
```

Actividad 13

Reproduce el ejemplo 11, añade comentarios que te ayuden a entenderlo y ejecútalo. Después:

- Modifica el programa de prueba para que accedan al almacén varios productores y varios consumidores. Podrás comprobar que no funciona correctamente. Averigua porqué y soluciona el problema.
- Prueba con diferentes variaciones del número de productores y consumidores y con diferentes valores para los tiempos de espera de ambos y analiza los resultados y comenta las conclusiones que saques de dicho análisis.

Actividad 14

Simular el uso del banco de un parque con un número determinado de plazas disponibles para que se sienten las personas que pasean por allí según las especificaciones siguientes:

- Todas las personas estarán en el parque desde el comienzo de la simulación e iniciarán su paseo inmediatamente.
- Cada una paseará durante un periodo de tiempo aleatorio entre 1000 y 3000 milisegundos. En este periodo se incluye el paseo y la llegada hasta el banco.
- Pasado el tiempo de paseo intentarán sentarse en el banco para descansar, momento en el que se dará una de las dos situaciones siguientes:
    - El banco tiene plazas libres, en cuyo caso se sentarán y descansarán durante un periodo de tiempo aleatorio entre 100 y 700 milisegundos. Transcurrido ese tiempo, se levantarán dejando libre su plaza.
    - El banco no tiene plazas libres, en cuyo caso esperarán hasta otra persona deje su plaza libre. Cuando eso ocurra, la obtención de esa plaza dependerá únicamente del carácter no determinista del planificador de la JVM.
- Se mostrarán en la consola mensajes que informen de cada nueva acción que emprende cada persona dentro de la simulación: pasea, espera una plaza libre en el banco, se sienta o se levanta.
- Antes de que comience la simulación, se pedirán por teclado el número de plazas del banco y el número de personas en el parque.

Opcional: desarrolla la aplicación con interfaz gráfica de usuario basada en Swing.

5 APIS de concurrencia de alto nivel

El uso de las APIs de concurrencia de bajo nivel, presentes en la plataforma Java desde sus comienzos, son adecuadas para problemas simples de concurrencia. Para problemas complejos, es preferible usar construcciones de más alto nivel, más fáciles de usar, menos propensas a que se cometan errores y más eficientes a la hora de explotar los recursos. Esto es especialmente cierto

en el desarrollo de aplicaciones que explotan de forma masiva la concurrencia en sistemas multiprocesador.

En esta sección, se exponen algunas de los recursos de alto nivel para el desarrollo de programas concurrentes, introducidos en el API de Java con la aparición de la versión 5.0 de la plataforma. Estos recursos se implementan en el paquete java.util.concurrent y se agrupan en:

- Locks (java.util.concurrent.locks): interfaces y clases para la gestión de bloqueos y esperas.
- Semáforos.
- Executors: interfaces y clases para la creación, ejecución y gestión de hilos.
- Variables atómicas (java.util.concurrent.atomic).
- Generación de números pseudoaleatorios de forma eficiente desde varios hilos con la clase ThreadLocalRandom.

También se han implementado nuevas estructuras de datos concurrentes en el *Java Collections Framework*.

5.1 El API de bloqueo

El API de bloqueo define una jerarquía de clases que tiene como objetivo proveer métodos de sincronización de hilos más flexibles y sofisticados que los proporcionados por los bloques sincronizados estándar. Entre ambas opciones existen algunas diferencias:

- Un bloque sincronizado está completamente contenido en un método, mientras que con la API de bloqueo es posible tener las operaciones de bloqueo y desbloqueo en métodos separados.
- Los bloques sincronizados no soportan la adquisición del bloqueo de forma equitativa, mientras que con la API de bloqueo esto se puede lograr estableciendo la propiedad de equidad a través de los constructores. Mediante esta propiedad se asegura que el hilo que ha permanecido más tiempo bloqueado obtenga el acceso al bloqueo antes que otros.
- Un hilo se bloquea si no puede obtener el acceso al bloque sincronizado, mientras que la API de bloqueo proporciona el método tryLock, que permite adquirir un bloqueo sólo si está disponible. Esto se traduce en la posibilidad de reducir el tiempo que un hilo permanece bloqueado.
- Un hilo que está en espera para adquirir el acceso a un bloque sincronizado no puede ser interrumpido, mientras que la API de bloqueo si permite hacerlo con el método lockInterruptably.

En la raíz de esta jerarquía de clases se encuentra la interface Lock que declara los métodos siguientes:

`void lock()`
Adquiere el bloqueo si está disponible. En caso contrario, el hilo permanecerá bloqueado hasta que lo adquiera.
`Void lockInterruptibly()`
Es similar al método *lock*, pero permite que un hilo bloqueado sea interrumpido para reanudar su ejecución.
`Boolean tryLock()`
Adquiere el bloqueo si está disponible retornando inmediatamente el valor *true*. En caso contrario, retornará inmediatamente el valor *false*.
`Boolean tryLock(long time, TimeUnit unit)`
Similar al anterior, excepto porque espera el tiempo especificado antes de dejar de intentar la adquisición del bloqueo para retornar el valor *false*.
`Void unlock()`
Libera el bloqueo.

5.1.1 Clase ReentrantLock

Implementa la interface Lock para ofrecer sincronización similar a la que se obtiene con recursos de bajo nivel, pero con capacidades extendidas.

El ejemplo siguiente muestra cómo usar esta clase para dar una solución alternativa a la expuesta en el ejemplo 10 para el problema del contador. Sólo se muestra la clase Contador, ya que la clase Main no se modifica.

Ejemplo 12

```
public class Contador {

  ReentrantLock lock = new ReentrantLock();
  private int n;

     public Contador(int n) {
         this.n = n;
     }

     public void inc() {
         lock.lock();
         try {
                // dentro del bloque try se recoge la sección crítica
                n++;
         } finally {
                lock.unlock();
         }
     }

     public int get() {
         return n;
     }
}
```

Es muy importante asegurarse de que las llamadas a métodos lock y unlock se ejecutan dentro de los bloques try y finally respectivamente, para evitar situaciones de interbloqueo.

El uso del método tryLock se resume en la plantilla siguiente:

```
public void tareaSincronizada () throws InterruptedException {
    //...
    boolean isLockAcquired = lock.tryLock(1, TimeUnit.SECONDS);

    if(isLockAcquired) {
        try {
            // section crítica
        } finally {
            lock.unlock();
        }
    }
    //...
}
```

El hilo que ejecute el método solicitará el bloqueo y si no lo obtiene durante el próximo segundo, abandonará la solicitud. Por tanto, sólo ejecutará la sección crítica si obtiene el bloqueo.

5.1.2 Clase Condition

Puede ocurrir, como quedó patente en el ejemplo de los productores/consumidores, que un hilo accede a una sección crítica y no puede avanzar hasta que se cumple una determinada condición.

La clase Condition, en conjunción con los bloqueos, permite implementar mecanismos de parada y avance dentro de una sección crítica. A continuación, se expone la solución al problema de los productores/consumidores usando estos recursos. De nuevo, sólo es necesario modificar la clase Almacen.

Ejemplo 13

```
public class Almacen {

    private Lock lock = new ReentrantLock();
    private Condition lleno = lock.newCondition();
    private Condition vacio = lock.newCondition();
    private String[] productos;
    private int stock = 0;

    public Almacen(int capacidad) {
        productos = new String[capacidad];
    }

    public void almacenar(String producto) {
        try {
            lock.lock();
            while (stock == productos.length)
                try { lleno.await(); } catch (InterruptedException e) {}
            productos[stock++] = producto;
        } finally {
            vacio.signalAll();
            lock.unlock();
        }
    }

    public String retirar() {
        try {
            lock.lock();
```

```
            while (stock == 0)
                try { vacio.await(); } catch (InterruptedException e) {}
            return productos[--stock];
        } finally {
            lleno.signalAll();
            lock.unlock();
        }
    }
}
```

5.1.3 Clase ReentrantReadWriteLock

Esta clase implementa la interface ReadWriteLock, que define la funcionalidad de un tipo de objetos que mantienen un par de bloqueos asociados, uno para operaciones de solo lectura y otro para operaciones de escritura. El bloqueo de escritura es exclusivo. El bloqueo de lectura puede ser retenido simultáneamente por múltiples hilos lectores, siempre que ningún hilo escritor haya obtenido el bloqueo de escritura.

Un bloqueo de lectura y escritura permite un mayor nivel de concurrencia en el acceso a datos compartidos que el permitido por un bloqueo de exclusión mutua. Aprovecha el hecho de que, si bien se anula la concurrencia cuando un hilo de escritura está modificando datos compartidos, en otro momento cualquier número de hilos de lectura puede leer los datos concurrentemente. El rendimiento aumenta de forma significativa cuando el tiempo total dedicado a lectura es mayor que el dedicado a escritura.

5.2 Semáforos

La clase Semaphore se usa para limitar el número de hilos que pueden acceder de forma concurrente a un recurso. Se basa en la gestión de un número determinado de permisos de acceso que los hilos deben adquirir cuando intentan acceder al recurso y liberar una vez hayan finalizado dicho acceso.

El constructor de esta clase acepta opcionalmente un parámetro booleano para determinar si se aplican criterios de equidad en la adquisición de los permisos. Cuando a este parámetro se le asigna el valor *true*, se aplica el criterio de equidad para evitar que un hilo irrumpa en la cola de espera y adquiera un permiso antes que otro que haya esperado más tiempo.

A continuación, se expone la solución al problema del banco del parque usando un semáforo.

Ejemplo 14

```
public class Parque {
    private static int NUMPERSONAS = 20;
    private static int NUMPLAZAS = 5;
    public final static Semaphore BANCO = new Semaphore(NUMPLAZAS);

    public static void main(String[] args) throws InterruptedException {
        Thread[] personas = new Thread[NUMPERSONAS];
        for (int i=1; i<NUMPERSONAS; i++)
            (personas[i] = new Persona(i + 1)).start();
        for (int i=1; i<NUMPERSONAS; i++)
            personas[i].join();
    }
}
```

```
public class Persona extends Thread{

 public Persona(int id) {
        super("Persona " + id);
```

```
  }

@Override
  public void run() {
    try {
      System.out.println(getName() + " pasea por el parque");
      Thread.sleep((long) (Math.random() * 2000 + 1000)); // tiempo paseando
      System.out.println(getName() + " llega al banco");
      if (Parque.BANCO.availablePermits() == 0)
          System.out.println("banco lleno, " + getName() + " espera");
      Parque.BANCO.acquire();
      System.out.println(getName() + " se ha sentado");
      Thread.sleep((long) (Math.random() * 500)); // tiempo sentada
      Parque.BANCO.release();
      System.out.println(getName() + " se ha levantado");
    } catch (InterruptedException e) {}
  }

}
```

Actividad 15

Reproduce los ejemplos 12, 13 y 14, añade los comentarios que te ayuden a entenderlos y ejecútalos.

Modifica el ejemplo 14 para solucionar los problemas siguientes:

- Si una persona llega al banco y no tiene sitio, volverá a pasear durante un tiempo aleatorio.
- Evitar que el lanzamiento de InterruptedException deje el permiso adquirido sin liberar.

Actividad 16

Cinco filósofos y filósofas de la antigua china, **孔夫子**, **楊朱**, **謝道韞**, **商鞅** y **吳諴真**, pasan su vida sentados alrededor de una mesa comiendo o pensando. A continuación, se proporciona una descripción detallada del proceso:

- Cada filósofo o filósofa tiene un plato de arroz que nunca se acaba, un palillo a la izquierda de su plato y otro a la derecha.
- Comparten los palillos a su izquierda y a su derecha con quienes se sientan a su izquierda y a su derecha respectivamente.
- En todo momento se encontrará en una de las situaciones siguientes:
    - Pensando: durante un periodo de tiempo aleatorio en el que no estarán usando ninguno de los palillos.
    - Esperando: desde el momento en que deciden comer, hasta que consiguen los dos palillos. Sólo podrán coger cada palillo si nadie lo está usando.
    - Comiendo: durante un periodo de tiempo aleatorio que comenzará inmediatamente después de que hayan cogido el segundo palillo. Cuando finalice este periodo de tiempo, dejarán los palillos libres.

Crear un programa Java que simule este proceso mediante hilos, uno por cada filósofo o filósofa, para que ejecuten las acciones de pensar, esperar o comer satisfaciendo la exclusión mutua sobre los palillos (dos no pueden emplear el mismo palillo a la vez). Además, se han de evitar el interbloqueo y la inanición. Desarrollar la aplicación con interfaz gráfica usando clases de la librería Swing y Java2D para mostrar una animación de la simulación con los elementos gráficos que se estimen oportunos, permitiendo la posibilidad de pausarla y reanudarla.

A continuación, se propone una forma de representar la escena en cada instante:

- Representar la mesa con un círculo.
- Representar cada plato mediante un círculo de un color que lo diferencie del resto y distribuirlos uniformemente alrededor de la mesa.
- Cuando un filósofo adquiere un palillo, dibujarlo del mismo color que su plato.
- Mostrar dentro de cada plato un icono que represente la acción que esté realizando el filósofo en cada instante: pensar, esperar o comer.

- Cuando un palillo no está siendo utilizado por ningún filósofo, se mostrará de un color diferente a cualquiera de los asignados a los platos.

5.3 Ejecutores

El uso de este tipo de recursos ayuda a separar, en aplicaciones de gran envergadura, la creación y gestión de hilos del resto de la aplicación.

El paquete java.util.concurrent define tres interfaces que determinan la funcionalidad básica los ejecutores:

- Executor: define el método execute para el lanzamiento de nuevas tareas. Siendo *r* un objeto Runnable y e un objeto Executor, la sentencia:

  ```
  (new Thread(r)).start();
  ```

  equivale a la sentencia:

  ```
  e.execute(r);
  ```

  Sin embargo, con los ejecutores, el momento en el que comienza la ejecución de la tarea depende de cada implementación de la interface Executor, siendo posible que no tenga lugar de forma tan inmediata como con el manejo de hilos a bajo nivel.
- ExecutorService, es una subinterface de la anterior que especifica la funcionalidad necesaria para el manejo del ciclo de vida, tanto de cada tarea individual, como del propio ejecutor. Complementa la funcionalidad del método execute con el método submit, sobrecargado para aceptar tanto objetos Runnable como objetos Callable y retornar objetos Future, que representan el resultado de una tarea asíncrona.
- ScheduledExecutorService, una subinterface de la anterior que ofrece la posibilidad de programar la ejecución de las tareas. Para ello define el método schedule también sobrecargado y que ejecuta tareas tras un tiempo de demora, y los métodos scheduleAtFixedRate and scheduleWithFixedDelay para ejecutar tareas que se repiten de forma periódica.

Normalmente, las referencias a ejecutores se declaran con una de estas interfaces.

5.3.1 Pools de hilos

La creación incontrolada de una cantidad elevada de hilos puede llevar al agotamiento de los recursos del sistema involucrados en esa tarea. Por otro lado, un aumento del paralelismo en un único procesador o núcleo de un procesador puede llevar a que el tiempo empleado en cada cambio de contexto resulte muy elevado en comparación con el que se asigna a cada hilo en cada turno de ejecución, haciendo que se pierda la ventaja del paralelismo. Ese podría ser el caso, por ejemplo, de un servidor web que cree un nuevo hilo para atender cada petición en un instante en el que el número de peticiones resulte muy elevado.

Los pools de hilos ayudan a prevenir el agotamiento de recursos evitando la creación incontrolada de nuevos hilos y a mantener el nivel de paralelismo en ciertos límites predefinidos. El servicio que proporcionan se basa en la creación de unos hilos denominados *worker threads*, que permiten su reutilización cuando finalizan la tarea asignada para ejecutar nuevas tareas. En el API de concurrencia de Java se implementan varios tipos de pools, cada uno con su propia estrategia para limitar el número de hilos activos, cuyo funcionamiento básico se describe a continuación:

- Cuando una aplicación necesita ejecutar una nueva tarea, se la pasa en forma de Runnable o Callable a un ExecutorService que implementa un pool de hilos para que la ejecute en un *worker thread*.

- El ejecutor busca en el pool un *worker threads* que haya finalizado su tarea. Si lo encuentra, lo reutilizará para ejecutar la nueva tarea. En caso contrario, se crea uno nuevo en el pool para ejecutarla, si la implementación del pool lo permite. En caso contrario se mantiene la tarea en espera.

La *factory class* Executors define varios métodos estáticos que retornan objetos ExecutorService que implementan diferentes tipos de pools:

```
static ExecutorService newCachedThreadPool()
```

Retorna un ejecutor que implementa un pool de hilos que crea nuevos *worker threads* sólo cuando ninguno de los que se hayan creado previamente esté disponible para ser reutilizado. Elimina de la cache aquellos hilos que permanezcan más de 60 segundos ociosos, evitando el consumo innecesario de recursos. Este tipo de pool se usa habitualmente en aplicaciones que ejecutan tareas asíncronas de corta duración.

```
static ExecutorService newFixedThreadPool(int nThreads)
```

Retorna un ejecutor que implementa un pool de hilos que crea a demanda un máximo de nThreads nuevos. Si el ejecutor recibe nuevas tareas cuando se ha creado la cantidad máxima de hilos y ninguno está disponible para su reutilización, estas permanecerán en espera en una cola de tareas.

```
static ScheduledExecutorService newScheduledThreadPool(int corePoolSize)
```

Retorna un ejecutor que implementa un pool de hilos y además permite programar la ejecución de las tareas. El parámetro corePoolSize determina el número máximo de *worker threads* que podrán permanecer ociosos en el pool esperando para ser reutilizados.

```
static ExecutorService newSingleThreadExecutor()
```

Retorna un ejecutor similar al retornado por newFixedThreadPool*(1)*, con la diferencia de que el retornado por newFixedThreadPool*()* no se puede reconfigurar para aumentar el número máximo de *worker threads* en el pool.

```
static ScheduledExecutorService newSingleThreadScheduledExecutor()
```

Similar al anterior, pero con la posibilidad de programar la ejecución de las tareas.

5.3.2 Asignación de tareas a un ExecutorService

Un ExecutorService puede ejecutar tareas Runnable y Callable asignadas usando alguno de los métodos siguientes:

- El método execute recibe un Runnable que representa la tarea a ejecutar, pero no da la posibilidad de obtener el resultado de la misma o comprobar su estado.
- El método submit recibe un Callable que representa la tarea a ejecutar y retorna un Future que proporciona métodos para comprobar si la tarea ha finalizado, para esperar a que finalice y para obtener el resultado de su ejecución.
- El método invokeAny asigna una colección de tareas al ejecutor y retorna el resultado de la ejecución de una de ellas que finalice de forma ordenada.
- El método invokeAll asigna una colección de tareas al ejecutor y retorna una lista de objetos Future.

5.3.3 Parada de un ExecutorService

Los métodos siguientes se usarán para gestionar la finalización ordenada del servicio:

- El método shutdown inicia la finalización del servicio. El servicio no finaliza de forma inmediata, sino que lo hará cuando hayan finalizado las tareas asignadas, pero entre tanto no se aceptarán nuevas tareas.
- El método shutdownNow intenta parar las tareas activas, bloquea el inicio de tareas que estuviesen en espera y retorna estas últimas en una lista.
- El método isShutdown retorna true si el servicio está en proceso de *shutdown*.
- El método isTerminated retorna true si previamente se ha invocado a uno de los dos métodos anteriores y el proceso de finalización del servicio se ha completado.
- El método awaitTermination hace que el hilo que lo invoca espere, durante un intervalo de tiempo especificado, a la terminación de todas las tareas asignadas a un ejecutor en proceso de *shutdown*. Retorna true si el servicio ha finalizado antes de agotar el tiempo especificado o false si se agotó el tiempo antes de la finalización de todas las tareas.