**Unidad 1. Programación concurrente**

1. **Introducción:**

* **Programa vs Proceso:**
* **Programa:** Conjunto de instrucciones almacenadas en un fichero en el disco duro.
* **Proceso:** Instancia de un programa en ejecución en memoria RAM o registros de la CPU.

**→ OJO:** Un programa puede lanzar varios procesos, por ejemplo, Chrome lanza un proceso por cada pestaña abierta.

* **Procesador vs núcleo:**
* **Procesador (CPU):** Componente físico del ordenador encargado de ejecutar las instrucciones de un proceso. A partir de 1971, el procesador se construye en único chip, por lo que se conocer como **microprocesador**.
* **Núcleo:** Cada uno de los procesadores que componen un microprocesador, que permiten ejecutar procesos de forma simultánea.
* **Monoproceso vs multiproceso:**
* **Sistemas monoproceso:** Solo permiten ejecutar un proceso a la vez (Windows antiguos).
* **Sistemas multiproceso:** Permiten varios procesos simultáneos a través de varios núcleos (CPUs modernas).
* **Monotarea vs multitarea:**
* **Sistemas monotarea:** No permite alternar entre procesos en ejecución. Cuando un proceso entra en ráfaga de E/S (espera un input), el procesador queda ocioso.
* **Sistemas multitarea:** Permite intercalar entre procesos en ejecución, optimizando su uso.

**→ OJO:** Prácticamente todos los sistemas actuales son multitarea y multiproceso.

* **Gestión de procesos en sistemas multitarea:**
* **Creación de procesos:** Al ejecutar el programa, carga las instrucciones en RAM y crea un registro denominado **Bloque de Control de Proceso (BCP)** que almacena toda la información relativa al proceso.
* **Eliminación de procesos:** Cuando la ejecución finaliza, el sistema operativo libera la memoria y recursos usados por el proceso.
* **Planificación de ejecución de procesos:** Decide qué proceso se ejecuta en cada momento (por ejemplo, en una ráfaga de E/S elige un proceso en espera), según el planificador de procesos.
* **Cambios de contexto:** Se encarga de desalojar el proceso actual, guardando sus datos en su BCP, y cargando un nuevo proceso con los datos de su BCP.
* **Asignación de recursos:** El sistema operativo se encarga de asignar la RAM, acceso a archivos y periféricos que precisa el programa.
* **Bloque de Control de Proceso (BCP):** Almacena la información relativa al proceso:
* **Identificador del proceso:** Número que identifica unívocamente el proceso.
* **Estado del proceso:** Ver siguiente apartado.
* **Contador de programa:** Dirección de memoria de la siguiente instrucción a ejecutar.
* **Registros de la CPU:** Registros internos del programa.
* **Información sobre gestión de memoria:** Recursos de la RAM destinados.
* **Información sobre E/S:** Dispositivos asignados, archivos abiertos, permisos, etc.
* **Estados de un proceso:**
* **Nuevo:** Acaba de ser creado.
* **Preparado:** Listo para ser ejecutado, a la espera de asignación de CPU por el sistema.
* **En ejecución:** Sus instrucciones se están ejecutando en la CPU.
* **Bloqueado:** No ha terminado su ejecución, pero se encuentra a la espera de un evento externo (por ejemplo, ráfaga de E/S).
* **Finalizado:** El programa ha terminado.

**A diagram of a diagram

Description automatically generated**

* **Hilos (*threads*):** Son procesos ligeros o subprocesos que permiten aligerar el trabajo de la CPU en cuanto a la gestión de un proceso. Son secuencias de instrucción de un mismo proceso, pero que pueden ejecutarse de forma alterna o simultánea. Como ocupan el mismo espacio de memoria, archivos y variables, el cambio de contexto y la comunicación entre ellos es mucho más sencilla que entre procesos.

**→** En la práctica, **los hilos permiten que un programa pueda hacer varias cosas a la vez**. Por ejemplo, que Spotify reproduzca música y permita buscar canciones a la vez de forma eficiente.

* **Programación concurrente:** Desarrollar programas cuyas partes se puedan ejecutar de forma desordenada mediante el uso de hilos. Como el sistema operativo no sabe qué tarea ejecutar en cada momento, es el propio programa (mediante **mecanismos de comunicación y sincronización** entre hilos) el que se encarga de ordenar la ejecución de las tareas. Ventajas:
* Mayor aprovechamiento de la CPU.
* Menor tiempo de ejecución.
* Facilita la resolución de problemas (por ejemplo, peticiones a servidor web de forma concurrente).

1. **Gestión de Hilos (I):**

**→ Recursos para la primera parte de la asignatura:**

* **IDE y lenguaje:** IntelliJ IDEA con lenguaje Java.
* **Repos:** Repos local y GitHub: <https://github.com/JJAS2022/Prog-de-Serv-y-Proc>
* **Creación de hilos:** Todo proceso tiene, al menos, un hilo al momento de crearse, el *main thread* o **hilo principal**, que es el que se lanza al ejecutar el método main en Java.
* **Clase Thread:** Gestiona la creación de hilos en Java.
* **Método sleep(milisegundos)**: Método estático de la clase Thread (no precisa objeto para ser llamado), que ordena al proceso que detenga la ejecución del hilo durante los milisegundos indicados, por ejemplo 2000 milisegundos son 2 segundos. Se emplea para simular las esperas que tiene que hacer un programa para realizar tareas complejas (por ejemplo, llamada a base de datos, proceso de escritura en disco, cálculos complejos, descarga de archivos, etc.). Una espera de 2 segundos es muchísimo en tiempo de procesamiento, por lo que se usa este método sleep() para simular ese comportamiento.

Thread.sleep(2000);

* **Excepción InterruptedException:** El método sleep() puede lanzar una excepción de tipo InterruptedException por lo que Java obliga a que se indique esto, que se puede hacer de dos formas:
* Con la palabra reservada **throws** en la declaración del método que lo invoca:

public static void prepararTostadas() throws InterruptedException {

Thread.sleep(2000);

}

* Encerrando el método sleep() en un **bloque try…catch**, que lo hace mucho más incómodo si hay que incluir varios métodos sleep(). En el método main sí que se incluiría el método prepararTostadas() dentro del bloque try…catch para capturar la excepción:

public static void prepararTostadas() {

try {

Thread.sleep(2000);

catch (InterruptedException ie) {

System.out.println(ie.getMessage())

}

}

* **Formas de crear hilos:**
* **Herencia de la clase Thread:**
* Para crear un hilo de la función prepararTostadas() en la clase Main, se crea una clase Tostadora que **extiende** (hereda de) la clase Thread:

class Tostadora extends Thread { }

* A continuación, dentro de la clase derivada, se **sobrescribe el método run()** de la clase padre Thread, incluyendo toda la funcionalidad del código que interesa que ejecute el hilo.

public void run() {  
 System.*out*.println("Tostadas: Comenzamos a preparar tostadas.");  
 System.*out*.println("Tostadas: Ponemos el pan a tostar.");  
 Thread.*sleep*(2000);  
 // Continúa…  
}

→ **Anotación @Override:** No cambia la funcionalidad del código, pero permite indicar al compilador que se ha sobrescrito el método de la clase padre para evitar problemas de compilación. Además, permite localizar rápidamente los métodos de la clase padre para sobrescribir el que nos interesa.

→ **Otra forma de sobreescritura rápida:** Dentro de la clase derivada, escribiendo el nombre del método de la clase padre, directamente se copia el texto que permite sobrescribirlo (por defecto, ejecuta el método del padre super.run()):

A screen shot of a computer program

Description automatically generated

* Sin embargo, este método no se puede compilar porque el método sleep() lanza una excepción que no está siendo capturada. Asimismo, no es posible capturarla con **throws** en la declaración del método run() porque al heredar de la clase Thread **no se puede modificar la firma o prototipo del método** (como tampoco podríamos añadirle parámetros o el tipo de retorno). En consecuencia, esto sí obliga a incluir un **bloque try…catch**:

public class Tostadora extends Thread{  
 @Override  
 public void run() {  
 try {  
 System.*out*.println("Tostadas: Comenzamos a preparar tostadas.");  
 System.*out*.println("Tostadas: Ponemos el pan a tostar.");  
 Thread.*sleep*(2000);  
 System.*out*.println("Tostadas: Echamos aceite.");  
 Thread.*sleep*(2000);  
 System.*out*.println("Tostadas: Echamos sal.");  
 Thread.*sleep*(2000);  
 System.*out*.println("Tostadas: Tostadas finalizadas.");  
 }catch (InterruptedException ie) {  
 System.*out*.println("Se ha interrumpido el hilo");  
 }  
 }  
}

* **Llamada del hilo desde la clase principal:** En la clase principal, se debe crear un objeto de la clase Tostadora (que es de tipo Thread porque hereda de esta clase) y lanzar su ejecución con el método start():

public class Main {  
 public static void main(String[] args) {  
 try {  
 Tostadora tostadora = new Tostadora();  
 tostadora.start();  
 *prepararCafe*();  
 } catch (InterruptedException ie) {  
 System.*out*.println("Hilo interrumpido");  
 }  
 }

}

→ **OJO:** El hilo se lanza con el método start(), no run(). Al lanzar el hilo, lo que va a hacer es ejecutar directamente la siguiente instrucción (prepararCafe()), y ambas funciones se van a ejecutar en paralelo (si no fuera un hilo, se habría ejecutado primero toda la tostada y luego todo el café):

A screen shot of a computer

Description automatically generated

* **Herencia de la interfaz Runnable:** El funcionamiento es muy parecido al anterior, pero en lugar de que la clase Tostadora extienda (herede de) la clase Thread, lo que hace es que implementa la interfaz Runnable.

→ **Interfaz:** Es muy similar a una clase, pero ninguno de sus métodos está implementado, sino que se deben desarrollar en la clase que la implementa.

→ **Interfaz Runnable:** Existe precisamente para que **los objetos que la implementen puedan ser ejecutados por instancias de la clase Thread**. Solamente tiene un método (sin implementar, obviamente), que se llama run().

* **Clase Tostadora que implementa Runnable y sobrescribe su método run():**

public class Tostadora implements Runnable {  
 @Override  
 public void run() {  
 try {  
 System.*out*.println("Tostadas: Comenzamos a preparar tostadas.");  
 System.*out*.println("Tostadas: Ponemos el pan a tostar.");  
 Thread.*sleep*(2000);  
 System.*out*.println("Tostadas: Echamos aceite.");  
 Thread.*sleep*(2000);  
 System.*out*.println("Tostadas: Echamos sal.");  
 Thread.*sleep*(2000);  
 System.*out*.println("Tostadas: Tostadas finalizadas.");  
 }catch (InterruptedException ie) {  
 System.*out*.println("Se ha interrumpido el hilo");  
 }  
 }  
}

* **Clase Main:** Aquí sí cambia, porque el objeto de tipo Tostadora no tendría el método start() que es el que permite ejecutar por hilos. Para poder hacerlo, primero hay que crear un objeto de tipo Thread, pasarle por parámetro un objeto que implemente la interfaz Runnable (basta con indicar new Tostadora, no hay que almacenar el objeto) y llamar al método start():

public class Main {  
 public static void main(String[] args) {  
 try {  
 Thread tostadora = new Thread(new Tostadora());  
 tostadora.start();  
 *prepararCafe*();  
 } catch (InterruptedException ie) {  
 System.*out*.println("Hilo interrumpido");  
 }  
 }  
}

→ **¿Es mejor usar Thread o Runnable?:** Es mejor usar la implementación de la interfaz Runnable y es la que se va a utilizar en el curso. Y ello porque si Tostadora hereda de la clase Thread no puede heredar de ninguna otra clase (solo puede heredar de 1 clase), por el contrario, si implementa la interfaz Runnable, puede heredar de otra clase e implementar otras interfaces.

* **Hilos con clases anónimas y funciones lambda:** Son formas de simplificar el código haciendo lo mismo de antes:
* **Usando una clase anónima que implementa la interfaz Runnable:** Esto permite no tener que crear una clase Tostadora aparte en otro archivo para poder ejecutar su método run().

→ **Clase anónima:** Es una clase que **no tiene nombre**, lo que evita tener que crear la clase en un archivo diferente, pero que **debe implementar una clase abstracta o una interfaz**. Es decir, es posible **definir e instanciar la clase desde el propio método que la invoca**. Declaración (**OJO:** lleva un “;” al final porque es una línea de código):

<Interfaz / Clase abstracta> objeto = new <Interfaz / Clase abstracta>() {

// Atributos y métodos de la clase anónima

}**;**

→ **OJO:** Puede parecer que lo que se esté haciendo sea instanciar un objeto de la interfaz o de una clase abstracta, lo cual es **imposible**. Por el contrario, lo que se está haciendo es **crear una clase sin nombre**, que implementa una interfaz o extiende una clase abstracta, con sus propios atributos y métodos, declarados en la propia instancia del objeto de esa clase.

→ **¿Cuándo utilizar una clase anónima?:** Si la clase solamente se va a utilizar una vez en todo el programa, seguramente lo mejor sería crearla como clase anónima. De lo contrario, si se va a utilizar varias veces, será mejor usar una clase normal, con su nombre e instancias.

→ **Accesibilidad de la clase anónima:** Las clases anónimas se crean dentro de otras clases, por lo que son clases anidadas. En este sentido, **las clase anónimas tienen acceso a los atributos y métodos de la clase dentro de la que se crean**.

* **Creación de la clase anónima dentro de la clase Main:** Como es una clase que implementa la interfaz Runnable, se crea llamando al constructor de la interfaz. Una vez creada, el propio IntelliJ permite añadir el método de la interfaz que hay que implementar (run()):

Runnable runnable = new Runnable() {  
 @Override  
 public void run() { }  
}

* **Añadir del código que sobrescribe el método dentro de la nueva clase anónima:**

Runnable runnable = new Runnable() {  
 @Override  
 public void run() {  
 try {  
 System.*out*.println("Tostadas: Comenzamos a preparar.”);  
 System.*out*.println("Tostadas: Tostamos el pan.”);  
 Thread.*sleep*(2000);  
 System.*out*.println("Tostadas: Echamos aceite.");  
 Thread.*sleep*(2000);  
 System.*out*.println("Tostadas: Echamos sal.");  
 Thread.*sleep*(2000);  
 System.*out*.println("Tostadas: Tostadas finalizadas.");  
 }catch (InterruptedException ie) {  
 System.*out*.println("Se ha interrumpido el hilo");  
 }  
 }  
}

* **Crear el objeto de la clase Thread que ejecutará el hilo, y pasarle como parámetros el objeto de la clase anónima que implementa Runnable:**

Thread tostadora = new Thread(runnable);

* **Lanzar el hilo con el método start():**

tostadora.start();

* **De hecho, es posible incluso simplificar aún más el código, sin necesidad de crear un objeto de la clase anónima, sino pasando directamente el constructor como parámetro del constructor del objeto de tipo Thread:**

Thread tostadora = new Thread(new Runnable() {  
 @Override  
 public void run() {  
 try {  
 System.*out*.println("Tostadas: Preparamos tostadas.");  
 System.*out*.println("Tostadas: Tostamos el pan.");  
 Thread.*sleep*(2000);  
 System.*out*.println("Tostadas: Echamos aceite.");  
 Thread.*sleep*(2000);  
 System.*out*.println("Tostadas: Echamos sal.");  
 Thread.*sleep*(2000);  
 System.*out*.println("Tostadas: Tostadas finalizadas.");  
 }catch (InterruptedException ie) {  
 System.*out*.println("Se ha interrumpido el hilo");  
 }  
 }  
});

* **Usando una función lambda:** En este caso, se utiliza una función lambda para sustituir la creación de una clase anónima, porque se implementa una interfaz (Runnable) con un único método que implementar (run()).

→ **Funciones lambda:** Son funciones que se pasan como parámetros de otras funciones, pero que no tienen nombre y se utilizan puntualmente en un momento del programa. Básicamente, funcionan igual que las funciones normales: tienen parámetros de entrada, hace cosas con ellos, y devuelve un valor.

* **Uso de la expresión lambda para sustituir la clase anónima:**

Thread tostadora = new Thread(() -> {  
 try {  
 System.*out*.println("Tostadas: Preparamos tostadas.");  
 System.*out*.println("Tostadas: Tostamos el pan.");  
 Thread.*sleep*(2000);  
 System.*out*.println("Tostadas: Echamos aceite.")

Thread.*sleep*(2000);  
 System.*out*.println("Tostadas: Echamos sal.");  
 Thread.*sleep*(2000);  
 System.*out*.println("Tostadas: Tostadas finalizadas.");

} catch (InterruptedException ie) {  
 System.*out*.println("Se ha interrumpido el hilo");  
 }  
});

* **Información sobre hilos:** Cada hilo tiene asociado una serie de información (atributos) que se puede obtener o modificar con **métodos de la clase Thread** (getters y setters):
* **getId():** Identificador único del hilo en la CPU. No se puede modificar por el programador.
* **getName():** Nombre del hilo. Se puede modificar con setName().
* **getPriority():** Prioridad de ejecución en la ejecución, entre 1 y 10. El planificador de procesos elige qué programa ejecutar según la prioridad indicada. Se puede indicar la prioridad con setPriority() indicando el número o bien un valor del enumerado Thread.MIN\_PRIORITY (valor 1), NORM\_PRIORITY (valor 5) y MAX\_PRIORITY (valor 10). Por defecto, la prioridad va a ser 5.
* **getState():** Indica el estado del hilo, no se puede modificar. Devuelve un enumerado con los valores posibles Thread.State.NEW, RUNNABLE, BLOCKED, WAITING, TERMINATED.
* **isAlive():** Valor booleano que indica si el proceso sigue ejecutándose.

→ **Hilo principal:** Para acceder a las propiedades del hilo principal, se crea un puntero de tipo Thread hacia el hilo actual (método estático currentThread()):

public static void main(String[] args) {  
 Thread thread = Thread.*currentThread*()  
 System.*out*.println("ID: " + thread.getId());  
 System.*out*.println("Nombre: " + thread.getName());  
 System.*out*.println("Prioridad: " + thread.getPriority());  
 System.*out*.println("Estado: " + thread.getState());  
 System.*out*.println("Está vivo: " + thread.isAlive());  
}

→ **Operatividad entre hilos:** Conocer la información de los hilos es útil para lanzar un hilo cuando haya finalizado otro, por ejemplo. Sin embargo, utilizar if durante la ejecución del hilo es inútil, porque no sabemos en qué momento se lanzará ese if. Mejor hacerlo con while y sleep, que comprueba la ejecución hasta que se produzca un cambio de estado:

while (thread.getState() != Thread.State.*TERMINATED*) {  
 try {  
 Thread.*sleep*(1000);  
 } catch (InterruptedException e) {  
 throw new RuntimeException(e);  
 }  
}  
System.*out*.println("Hilo terminado.");

Alternativamente, puede usarse isAlive en lugar de getState():

while (thread.isAlive()){}

Sin embargo, estas formas de esperar a que un hilo termine no son las mejores, porque el tiempo de espera puede ser demasiado grande o pequeño, y emplear CPU continuamente en esa comprobación. Más adelante, se usará el método **join()** que es el adecuado.

→ **Gestionar varios hilos:**

* Generar n hilos (según input del usuario) con un bucle for, dándoles de nombre el valor de la variable i + 1:

System.*out*.print("Introduce el número de hilos a crear: ");  
while (!*scanner*.hasNextInt()) {  
 *scanner*.nextLine();  
}  
int n = *scanner*.nextInt();  
  
for (int i = 0; i < n; i++) {  
 Primos primos = new Primos();  
 primos.name = String.*valueOf*(i+1);  
 Thread thread = new Thread(primos);  
 *threads*.add(thread);  
 thread.setName(primos.name);  
 System.*out*.println("Hilo " + thread.getName() + ": Mostrando primos hasta el " + primos.maxValue);  
 thread.start();  
}

* Añadirlos a un ArrayList que almacene objetos de tipo Thread, y acceder a ellos con un método específico para comprobar si están todos vivos:

static ArrayList<Thread> *threads*= new ArrayList<Thread>();

*threads*.add(thread);

private static boolean threadsAlive() {  
 for (int i = 0; i < *threads*.size(); i++) {  
 Thread thread = *threads*.get(i);  
 if (thread.isAlive())  
 return true;  
 }  
 return false;  
}

* Mientras haya al menos un hilo vivo, el hilo principal va a ir imprimiendo el ID, nombre y estado de cada hilo almacenado en el ArrayList, y esperar 1 segundo para volver a hacer la comprobación:

while (*threadsAlive*()) {  
 for (int i = 0; i < n; i++) {  
 Thread subThread = *threads*.get(i);  
 System.*out*.println("ID: " + subThread.getId() + ", Hilo: " + subThread.getName() + ", Estado: " + subThread.getState());  
 }  
 try {  
  
 Thread.*sleep*(1000);  
 } catch (InterruptedException e) {  
 throw new RuntimeException(e);  
 }  
}

1. **Gestión de Hilos (II):**

* **Suspensión de hilos:**
* **Concepto:** Detener de forma temporal un hilo, por múltiples motivos (por ejemplo, a la espera de una condición, a la espera de que otro hilo finalice, hacer consultas al servidor cada cierto tiempo, etc.).
* **Formas de suspender un hilo:**
* **Método estático sleep(milisegundos) de la clase Thread:** Detiene su ejecución durante los milisegundos indicados.

Thread.sleep(2000); // Suspende durante 2 segundos

* **Clase TimeUnit:** Contiene una serie de constantes que facilitan indicar la cantidad de tiempo que se debe suspender el hilo (DAYS, HOURS, MINUTES, SECONDS, MILISECONDS). Se emplea junto con el método sleep():

TimeUnit.SECONDS.sleep(2); // También suspende durante 2 segundos

**→ Ventaja:** Es más flexible que solo usar sleep(), porque evita tener que calcular, por ejemplo, cuantos milisegundos son 1 minutos.

* **Método estático yield() de la clase Thread:** Este método indica al compilador que el hilo está listo para ser suspendido porque no está haciendo nada, pero no lo suspende automáticamente como sleep(). Da paso a otros hilos que sí tienen ejecución pendiente.
* **Método join():** Permite **suspender un hilo hasta que finalice la ejecución de otro**. En el hilo principal, por ejemplo, se vincula el hilo secundario con el método join(), de forma que queda a la espera de que aquel finalice para continuar su ejecución:

public class Main {  
 public static void main(String [] args) throws InterruptedException {  
 Thread thread = new Thread( () -> {  
 try {  
 Random random = new Random();  
 for (int i = 0; i <10; i++) {  
 Thread.*sleep*(random.nextInt(1500));  
 System.*out*.println("Número " + i);  
 }  
 } catch (InterruptedException ie){  
 System.*out*.println("El hilo se ha interrumpido.");  
 }  
 });  
 thread.start();  
 thread.join();  
 System.*out*.println("El hilo ha finalizado");  
 }  
}

**→ Llamada bloqueante:** En el momento en que se llama al método join(), el hilo que lo invoca queda bloqueado a la espera de que termine el hilo llamado.

**→ Espera activa:** Emplear un *while(hilo.isAlive()) {}* se conoce como espera activa, pues está constantemente preguntando si un hilo ha terminado, lo que supone un enorme gasto de CPU. Por ello, no es nada eficiente comparando con el método join().

* **Método wait():** También permite suspender la ejecución de un hilo, dejándolo bloqueado hasta que recibe una señal de continuar. Se ve más adelante cuando se estudia la sincronización de hilos.
* **Finalización de hilos:**
* **Concepto:** Hasta ahora, se ha visto que un hilo finaliza cuando termina toda su ejecución. Sin embargo, hay hilos de tareas repetitivas que no se sabe cuándo van a terminar (por ejemplo, durante una videollamada, se envían y reciben señales en un hilo, pero mientras tanto el mismo programa permite enviar mensajes de texto en otro hilo), de modo que el hilo entra en un bucle infinito que se debe finalizar.
* **Formas de detener hilos:**
* **Método stop():** Era la forma antigua de finalizar hilos, pero hoy está desaconsejado por problemas de seguridad.
* **Mediante de flags:** Una **flag** no es más que una variable que permite modificar el comportamiento de un bucle, en este caso, un valor booleano. Básicamente, consiste en cambiar el valor del booleano para ordenar que el bucle se detenga, por ejemplo:

public class Cafetera implements Runnable{  
 private int contador = 0;  
 private boolean ejecutar = true;  
  
 public int getContador() {  
 return contador;  
 }  
  
 public void Detener() {  
 ejecutar = false;  
 }  
  
 @Override  
 public void run() {  
 try {  
 while(ejecutar) {  
 Thread.*sleep*(200);  
 Thread.*sleep*(200);  
 Thread.*sleep*(200);  
 contador++;  
 System.*out*.println("Número de cafés preparados: " + contador);  
 }  
 } catch (InterruptedException ie) {  
 System.*out*.println("Hilo interrumpido.");  
 }  
 }  
}

En este caso, el bucle while() se mantiene activo mientras el booleano ejecutar sea true. La llamada al método Detener() modifica el valor del booleano a false y se podría salir del bucle. Sin embargo, el bucle ejecutará completamente la instrucción actual antes de salir.

Por su parte, en el método main, haríamos la llamada al hilo, que entra en el bucle, y da la posibilidad al usuario de detenerlo pulsando Enter. Esta acción llama a la acción Detener() y finaliza el hilo cuando termine la actual ejecución del bucle.

public class Main {  
 public static void main(String[] args){  
 Cafetera cafetera = new Cafetera();  
 Thread thread = new Thread(cafetera);  
 thread.start();  
  
 System.*out*.println("Presione intro para detener.”);  
 Scanner scanner = new Scanner(System.*in*);  
 scanner.nextLine();  
 cafetera.Detener();  
  
 try {  
 thread.join();  
 } catch (InterruptedException ie){  
 ie.printStackTrace();  
 }  
  
 System.*out*.printf("Total: %d cafés",

cafetera.getContador());  
 }  
}

**→ OJO:** Es muy importante incluir el thread.join() en el método main (hilo principal) para que espere a que el hilo secundario termine su ejecución antes de continuar. De lo contrario, arrojaría datos erróneos, sin que el hilo secundario haya terminado aún su ejecución.

* **Mediante interrupciones:** Si bien el uso de flags para detener hilos es común a todos los lenguajes de programación, el uso de interrupciones es **exclusivo de Java**. Para usar interrupciones, se debe llamar al **método interrupt() de la clase Thread**:

thread.interrupt();

**→ Excepción InterruptedException:** Cuando **un hilo está en un** **estado de WAITING** (por haberse lanzado un método sleep(), join() o wait()), **y se lanza el método interrupt()**, se producirá esta excepción. Por ello, siempre que se incluyen estos métodos, se debe incluir un *throws InterruptedException* en el método que lo llama, o encapsularlo en un *bloque try…catch*.

**→ Bucle:** Dado que ya no se va a hacer con una flag, no es necesario el booleano (e.g., ejecutar) no el método que cambiaba su valor (e.g., Detener()). Sin embargo, el bucle infinito sigue siendo necesario para las continuas llamadas del hilo (un *while(true)* es una muy maña práctica). En su lugar, se utiliza un método de la clase Thread que permite verificar si un hilo ha sido interrumpido (**isInterrupted()**) y detener la ejecución cuando sea true para la instancia del hilo actual (currentThread()):

while (!Thread.currentThread().isInterrupted()) {}

**→ Gestión de la excepción:** En este caso, si se interrumpe durante el estado WAITING, no se termina de ejecutar el bucle while, sino que sale directamente hacia el bloque catch. Si, por el contrario, se lanza la interrupción sin que el programa esté WAITING (por ejemplo, porque no haya ningún sleep, join o wait), el bucle no se interrumpe porque no hay bloque try…catch, pero a la siguiente pasada ya no se ejecuta.

**→ OJO:** En el ejercicio de práctica, tuve que poner dos scanner.nextLine() para que no saltara la interrupción directamente.

1. **Sincronización de hilos:**

* **Objetivo:** Que varios hilos compartan memoria facilita que se comuniquen y puedan acceder a los mismos datos. Esto es muy útil pero puede ser peligroso porque compartir memoria puede dar **problemas de concurrencia** (condiciones de carrera y secciones críticas).
* **Ejemplo 1:** En primer lugar, se crea un objeto Contador (básicamente cuenta hasta 100) y se lanza en un hilo. Como está en un bucle for, se crearán 5 contadores y 5 hilos. Este código no comparte memoria y es poco eficiente. Cada hilo devolverá 100.

for (int i = 1; i <= 5; i++) {

Contador c = new Contador();  
 Thread thread = new Thread(c);  
 thread.start();  
}

* **Ejemplo 2:** En este caso, se crea un único objeto Contador fuera del bucle, y 5 hilos que trabajan con el mismo objeto (misma dirección de memoria, atributos, etc.). En este sentido, si bien cada hilo cuenta hasta 100, lo hace sobre la misma variable del objeto, de modo que un hilo mostrará 100, otro 200, y así hasta 500. Si bien es más eficiente, es muy **peligroso**, porque no es el resultado esperado.

Contador c = new Contador();  
for (int i = 1; i <= 5; i++) {  
 Thread thread = new Thread(c);  
 thread.setName("Hilo " + i);  
 thread.start();  
}

* **Condición de carrera:** Comportamiento del software por el que el resultado que se obtiene depende del orden en el que se ejecutan las operaciones concurrentes (hilos). Es decir, **si al cambiar el orden de ejecución de los hilos se producen resultados diferentes**, estamos ante una condición de carrera. Ello implica que el resultado no es predecible, debido al uso de recursos compartidos a los que tienen acceso varios hilos.
* **Sección crítica:** Partes del código que acceden y modifican un recurso compartido.
* **Monitores:** Estructuras diseñadas para impedir que más de un hilo acceda a la sección crítica de un objeto, sincronizando el acceso de los distintos hilos. Cualquier objeto de Java puede actuar como monitor de sus propias secciones críticas y las de otros objetos, ya que todos heredan de la clase Object.
* **Problema:** Clase Cuenta, tiene un atributo saldo, y dos métodos, ingresar() que aumenta el saldo, y reintegrar() que lo reduce. Los métodos ingresar() y reintegrar() constituyen las secciones críticas porque acceden a un recurso compartido: la variable saldo.
* **Solución:** Crear un monitor en la propia clase Cuenta que controle el acceso a sus secciones críticas. El monitor tiene tres partes:
* **Entry set:** Los hilos que quieran acceder a una sección crítica entran por aquí (**entry**) y se verifica si hay algún otro hilo ejecutando una sección crítica.
* **Owner:** Si no hay ningún hilo ejecutando la sección crítica, el hilo que ha accedido al Entry set se convierte en el propietario del monitor (**acquire**) y podrá ejecutar la sección crítica. Hasta que ese hilo no libere la sección crítica (**release**), no se permitirá acceder a ningún otro hilo.
* **Wait Set:** Los hilos que se suspenden con el método wait() pasan aquí y ceden la propiedad u otro hilo, a la espera de que se lance un método notify() que lo saque de este estado.
* **¿Cómo se crean los monitores?:** No se crean, Java los crea automáticamente cuando se indica, mediante métodos o bloques sincronizados, que hay una sección crítica que proteger en algún recurso.
* **Mecanismos de sincronización de hilos:**
* **Métodos sincronizados:** Impiden que otro hilo acceda a c**ualquiera de los métodos sincronizados de un objeto** mientras esté alguno de ellos en ejecución. Esto garantiza que siempre se ejecuten de forma sincronizada. Se declaran con la palabra reservada **synchronized** en la declaración del método:

class Cuenta{

private int saldo;

public synchronized void ingresar(int cantidad) {

saldo += cantidad;}

public synchronized void reintegrar(int cantidad) {

saldo -= cantidad;}

}

**→ Funcionamiento:** Ambos métodos de la clase son mutuamente excluyentes, mientras uno de ellos se ejecuta, el otro no puede ejecutarse.

* **Bloques sincronizados:** Se emplean cuando no se tiene acceso al código fuente de la función que ejecuta acciones de sección crítica (por ejemplo, porque es una librería externa). En este caso, se crea un bloque que sólo puede ser ejecutado por un hilo a la vez, pasándole como parámetro el objeto que actuará como monitor:

synchronized (<objeto>) {

// Instrucciones a sincronizar

}

**→ Ejemplo:** Si no fuera posible acceder a la clase Cuenta, se debería añadir los bloques sincronizados en las funciones que hacen uso de sus secciones críticas:

private final Cuenta cuenta;

public void run() {

for (int i = 0; i < 1000; i++) {

synchronized(cuenta) {

cuenta.ingresar(1000);

}

}

}

**→ OJO:** Si bien cualquier objeto puede actuar como monitor, se debe incluir **el mismo objeto en todos los bloques que quieran ser sincronizados**. Es decir, no puede ser que cada hilo tenga su instancia de una clase, sino que el objeto debe ser compartido entre todos los hilos. En caso de que no exista un objeto similar en el código (en el ejemplo anterior sí existe, el objeto *cuenta*), es posible crear un objeto de tipo Object que haga esta función:

Cuenta cuenta = new Cuenta();

Object object = new Object();

HiloIngreso hi = new HiloIngreso(cuenta, object);

HiloReintegro hr = new HiloReintegro(cuenta, object);

* **Métodos notify() y wait():** Son métodos de la clase Object, de modo que todos los objetos lo tienen y forman parte de los monitores. **Cuando se llama al método wait() de un objeto, el hilo que lo ejecuta se suspende hasta que se llama al método notify() sobre ese mismo objeto**. A tener en cuenta:
* Los métodos wait() y notify() siempre hay que **usarlos dentro de métodos o bloques sincronizados**, o darán error.
* Los métodos notify() y wait() hay que **lanzarlos desde el mismo objeto**. De lo contrario, se estarían usando monitores distintos.
* La condición para lanzar el método wait() (en este caso, que el saldo es positivo) **se debe comprobar siempre en un bucle while** para evitar “*spurious wakeups*” o falsos positivos. De este modo, al recibir la notificación se volverá a comprobar en el bucle la condición y se evita ejecutar el hilo suspendido con una condición que no se cumple.
* El envío del aviso con notify() **no implica que el hilo suspendido vaya a continuar en ese momento**, sino que indica que ya puede seguir, pero es el SO el que decide cuándo se retoma.
* El método notify() avisa al objeto de que se puede retomar la actividad del hilo que estaba suspendido. Si hay varios suspendidos, se tomará uno aleatoriamente, si se quieren despertar todos, se puede hacer con el método **notifyAll**().

**→ Ejemplo:** Para el ejemplo de la cuenta bancaria, se pone un wait() en el método de reintegrar() para que se detenga mientras no haya saldo en la cuenta. Por su parte, se pone un notify() en el método de ingresar() para que la cuenta sepa que ya puede dar dinero porque hay saldo.

synchronized (cuenta) {

while (saldo == 0) {

cuenta.wait();

}

cuenta.reintegrar();

}

synchronized (cuenta) {

cuenta.ingresar();

cuenta.notify();

}

**→ OJO:** El uso de un notify() dentro de un bucle while puede dar problemas porque el bucle (si está sincronizado) monopoliza el monitor y nunca da paso al hilo notificado. Par evitarlo, conviene sincronizar solo la parte que accede a la sección crítica en lugar de todo el bucle. Por ejemplo:

private void producirCapsula() throws InterruptedException {  
 while (contenedor.getCapsulas() < 100) {  
 Capsula capsula = new Capsula(variedad, intensidad);  
 synchronized (contenedor) {  
 contenedor.anyadirCapsula();  
 System.*out*.println("Hilo Productor " + Thread.*currentThread*().getName() + ": Se ha fabricado una cápsula. Total en contenedor: " + contenedor.getCapsulas());  
 contenedor.notify();  
 }  
 Thread.*sleep*(random.nextInt(500, 1000));  
 }  
}

1. **Gestión de múltiples hilos:**

* **ExecutorService:** Interfaz que facilita la creación y gestión eficiente de hilos en Java. Esto permite que un hilo ejecute varias tareas, o sea, que se pueda reutilizar creando un *pool* de hilos (conjunto de hilos) que se pueden reutilizar. En este sentido, esta interfaz permite crear varios hilos que estén disponibles para realizar tareas y se asignen automáticamente. Si todos los hilos están ocupados, se detendrá la asignación hasta que se desocupe alguno, pero no se destruye ni se crean nuevos hilos. Esta interfaz tiene dos métodos:
* **void execute (Runnable command):** Ejecuta en un hilo la tarea que recibe por parámetro, que debe ser de tipo Runnable.
* **void shutdown():** Impide que entren nuevas tareas al ejecutor, pero permite que las tareas en marcha y en cola se ejecuten hasta su finalización normal.
* **List<Runnable> shutdownNow():** Finaliza las tareas en ejecución de forma abrupta y devuelve una lista con las tareas en cola.
* **boolean awaitTermination(long timeout, TimeUnit unit):** Bloquea el hilo a la espera de que finalicen sus tareas o el tiempo que recibe por parámetro. Se usa junto con shutdown().
* **ThreadPoolExecutor:** Como la interfaz ExecutorService no se puede instanciar, se hace a través de dos clases: **ThreadPoolExecutor y ScheduledThreadPoolExecutor**. Estas clases permiten ejecutar los hilos disponibles en el pool, y son muy configurables a través de parámetros (número de hilos, si se crean a petición o por defecto, tiempo de vida, etc.). La única diferencia es que Scheduled permite realizar tareas de forma programada. Métodos útiles:
* **public int getPoolSize():** Devuelve el número de hilos en el pool.
* **public int getActiveCount():** Devuelve el número de hilos en ejecución.
* **Executors:** Es una clase que permite crear *pools* de forma sencilla utilizando métodos estáticos:
* **static ExecutorService newFixedThreadPool (int nHilos):** Recibe por parámetros el número de hilos a crear y devuelve un objeto ThreadPoolExecutor de tamaño fijo nHilos. Ese objeto luego puede lanzar objetos que implementen la interfaz Runnable y organizar como se van a ir ejecutando según los hilos de los que dispone para adjudicar. Por ejemplo:

ExecutorService Executor = new Executors.newFixedThreadPool(2);

for (int i = 0; i < 10; i++) {

Contador c = new Contador();

executor.execute(c);

}

executor.shutdown();

* **static ExecutorService newSingleThreadExecutor():** No tiene parámetro porque crea un pool con un único hilo que ejecuta una cola de tareas una a una
* **static ExecutorService newCachedThreadExecutor():** No recibe parámetro porque crea un *pool* de tamaño variable, de modo que los hilos se crearán según se necesiten. Los hilos se mantienen vivos durante 60 segundos tras finalizar su tarea y, si no reciben una nueva, se eliminan del *pool*.
* **static ExecutorService newScheduledThreadExecutor(int nHilos):** Recibe por parámetro el número de hilos a crear y devuelve un objeto ScheduledExecutorService, que dispone de varios métodos para lanzar los hilos con retraso o periódicamente.
* **static ExecutorService newSingleThreadScheduledExecutor():** Similar al anterior, pero solamente crea un hilo no configurable.