UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

FACULTAD DE CIENCIAS

# Práctica 1 : Introducción a hilos


ALUMNO

**Javier Alejandro Rivera Zavala -** `311288876`

PROFESOR TITULAR

**Gilde Valeria Rodríguez Jiménez**

PROFESORES ADJUNTOS

**Rogelio Alcantar Arenas**
**Gibran Aguilar Zuñiga**
**Luis Ángel Leyva Castillo**


ASIGNATURA

**Computación concurrente**

*14 de Febrero del 2024*

## Preguntas

1. ¿Por qué se utiliza Interrupted Exception en el método main?
   Sirve para manejar la excepción que podría ocurrir cuando se utiliza el método join() para esperar a que los hilos terminen. El método join puede provocar que se lance la excepción InterruptedException si el hilo actual es interrumpido mientras está esperando a que otro hilo termine su ejecución.

2. ¿Para qué sirve el método Join?
   En este caso, sirve para que el hilo que manda a llamar a join(), digamos h1, espere a que el hilo desde donde se llama dicho método, el de main, termine su ejecución. Sirve para "frenar" un hilo y que ya no se ejecute nada dentro de él.

3. ¿Qué pasa si no le hacemos Join a los hilos?
   El hilo que mande a llamar a join() dentro de otro, no esperara a que el hilo desde el cuál se llamo a join() termine su ejecución y por lo tanto se mantendran en ejecución de forma concurrente

4. ¿Cuales son las ventajas de implementar Runnable contra extender de Thread?
   Implementar una interfaz siempre es preferible antes que extender una clase, toda vez que Java así como otros lenguajes de programación, maneja un sistema de herencia en una sola línea, lo cuál limita el comportamiento y atributos de que podemos añadir a la clase que extiende, en comparación, una clase puede implementar cuantas interfaces quiera y así verse mucho más enriquecida.

5. ¿Cuál es la diferencia de implementar Runnable contra Callable?
   Callable es una versión mejorada de Runnable, podemos ejecutar tareas de tipo Runnable haciendo uso de la clase Thread o ExecutorService, pero sólo podemos emplear Callable con ExecutorService. La interfaz Runnable sólo cuenta con el método run() que no tiene excepciones verificadas ni devuelve ninguna clase de valor/objeto, por otro lado, Callable cuenta con call() que si puede lanzr excepciones verificadas y devolver valores.

6. ¿Se puede predecir el orden en el que se imprime el mensaje de la clase Hilos?
   No, pueden aparecer el cualquier orden.

7. En el archivo Hilos2.java, ¿Qué pasa si sacamos la instancia de la clase "h" de t1, es decir, poner h antes de declarar t1?
   El objeto de la clase Hilos2 podría vivir fuera del hilo hasta que la memoria correspondiente al mismo, sea desasignada, en cambio, con la implementación actual, h sólo vive dentro de t1.

8. ¿Cómo podríamos darle un comportamiento diferente a los hilos?
   Una posible forma de modificar el comportamiento de los hilos de forma individualizada es crear los hilos como en el archivo Hilos2 y sobreescribiendo el método run() de cada hilo de forma distinta, esto nos permitiría dotar de "individualidad" a cada hilo.

9. **Sección Synchronized:** Antes de empezar, piensa que es lo que hace Synchronized, analizando lo que paso en los ejercicio anteriores, ¿Que significara?...
   Posterior a eso, realiza una breve investigación de lo que hace y escríbelo de forma breve. Descomenta el siguiente bloque de código y comenta el que no lo estaba, ¿Notas alguna diferencia?
   Synchronized sirve, como cabría esperar por su nombre, para sincronizar el acceso a un recurso compartido por parte de los hilos, de modo que sólo un hilo pueda acceder y modificar uno de esos recursos a la vez. Es importante hacer notar que, aunque para esta práctica nos centramos en los hilos, existe también la sincronización de procesos. En java la sincronización se implementa a través de locks (candados ó cerraduras), sólo un hilo puede tener uno de dichos locks en un determinado momento, lo que se conoce como estar en el monitor, situación que sólo es posible para uno y sólo un hilo a la vez. La sincronización permite la exclusión mutua, de modo que un hilo no puede intervenir con las tareas del otro y viceversa, en lo que se refiere al acceso de recursos compartidos.

10. Escribe que variables son locales (variables que están en memoria del hilo) y que variables son compartidas de cada archivo y el por qué. Puedes tomarle captura al código y encerrar en un recuadro dichas variables.

- **Clase Contador:** Cada hilo cuenta con un nombre propio que a su vez le sirve como identificador idHilo y por lo tanto es una variable del hilo, mientras que el número de rondas es una "constante" global pues es de clase. El valor es una variable compartida y el arreglo para permisos de mi implementación también es accesible para todos los hilos, aunque guarda permisos asignados a cada hilo individual, pero de forma externa. También viven en el hilo numHilos, i e incluso e.

```
public static final int RONDAS = 10000;
private int valor;
private volatile boolean[] permisoHilos;
```

```
@Override
public void run() {
    System.out.println("DENTRO RUN");
    suma();
}

/**
* Método que realiza la operación de suma controlada por el mecanismo de permisos.
*/
public void suma() {
    int numHilos = permisoHilos.length;
    int idHilo = Integer.parseInt(Thread.currentThread().getName());

    for (int i = 0; i < RONDAS; ++i) {
        try {
            Thread.sleep(1); // Simulación de una operación que requiere tiempo
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        while (!permisoHilos[idHilo - 1]) {/*Esperando*/}

        valor += 1;

        // Actualiza los permisos para el próximo hilo
        permisoHilos[idHilo - 1] = false;
        permisoHilos[(idHilo % numHilos)] = true;
    }
}
```

- **Clase ContadorC:** La variable contador es accesible para todos los hilos así como las constantes RONDAS e HILOS pues se definen como atributos de clase, por otro lado, las variables total e i viven dentro del hilo ya que son locales del método call().

```
public static Integer contador = 0;
public static final Integer RONDAS = 10000;//Cambiar esta variable para jugar con los valores. USar Thread Sleep
public static final Integer HILOS = 4;

@Override
public Integer call() throws Exception {//Esta metodo se implementa, retornamos el valor del Generico
    System.out.println("INCIO CALL");
    int total = 0;
    for(int i = 0; i< RONDAS; ++i){//Aqui tienen que hacer lo que dice en el pdf, objetivo llegar a los valores 2
        total += 1;
        contador += 1;//Si da un error de Sonar ignorarlo, por el momento, es más didactico
    }
    return total;//Realmente esta variable es para que descubran cosas :O
}
```

- **Clase ContadorS:** Nuevamente, RONDAS es una constante de clase y por lo tanto accesible a todos los hilos, lo mismo para la variable valor que es un recurso compartido a modificar por cada hilo al llamar suma. Otra variable local para cada hilo es i que sólo vive en el for de suma, en cualquiera de sus implementaciones.

```
private static final int RONDAS = 10000;
private int valor;
```

```
@Override
public void run() {
    suma();
}
/*
public synchronized void suma(){
    for(int i = 0; i < RONDAS; ++i){
        valor = valor + 1;
    }
}
*/

public void suma(){
    synchronized(this){
        for(int i = 0; i < RONDAS; ++i){
            valor = valor + 1;
        }
    }
}
```

- **Clase herencia/Hilos:** La variable contador es accesible a todos los hilos al estar declarada de forma global como variable de clase, por otro lado, las constantes HILOS y RONDAS son también de clase y por lo tanto accesibles para todos los hilos. Como variables de hilo tenemos id_hilo pues cada hilo cuenta con un identificador propio y las variables i junto con exception son locales del método suma por lo que sólo viven dentro del hilo.

```
public static final Integer HILOS = 3;
public static final Integer RONDAS = 10000;
private static int contador = 0;
private int id_hilo;
```

```java
@Override
public void run() {
    System.out.println("Hilo " + id_hilo + " corriendo");
    suma();
}

/**
* Método que realiza la suma controlada por el hilo, mediante rondas de incremento y sincronización.
*/
public void suma() {
    for (int i = 0; i < RONDAS; ++i) {
        try {
            Thread.sleep(1); // Simulación de una operación que requiere tiempo
        } catch (InterruptedException exception) {
            exception.printStackTrace();
        }

        // Sincronización para que cada hilo incremente el contador en su turno
        while (i % HILOS != id_hilo - 1) {
            try {
                Thread.sleep(1);
            } catch (InterruptedException exception1) {
                exception1.printStackTrace();
            }
        }

        contador += 1;

        // Cede el control del procesador a otros hilos
        Thread.yield();
    }
}
```

- **Clase hilos/Hilos:** En este caso, todas nuestras variables viven dentro de un hilo, existe la variable "hilos" que vive dentro del hilo principal y que apunta a la estructura de datos en la que están almacenados el resto de hilos. En las clases anteriores omití las variables locales del hilo principal en main, pues resultaba repetitivo y no es lo que se esperaba dado el contexto de la pregunta, sin embargo, hago notar que en todas ellas las variables locales de main, pueden considerarse como variables de hilo.

```java
@Override
public void run() { //Sobrescribimos el metodo run
    int a = 10;
    int b = 12;
    int ID = Integer.parseInt(Thread.currentThread().getName());
    if(ID == 1){
        System.out.println("Soy el hilo 1");
    }else{
        System.out.println("Hola soy el hilo: "+ Thread.currentThread().getName());//Pedimos el nombre del hilo pidiendo
    }
}
```

```java
public static void main(String[] args) throws InterruptedException {
    Hilos h = new Hilos();//Se crea una instancia de la clase
    Thread t1 = new Thread(h,"1");//Creamos un hilo, le pasamos de parametro la instancia de la clase y un nombre
    Thread t2 = new Thread(h,"2");
    Thread t3 = new Thread(h,"25");
    Thread t4 = new Thread(h,"45");

    ArrayList<Thread> hilos = new ArrayList<Thread>();       //Estructura para guardar 10 hilos

    for(int i=0; i< 10; i++){
        hilos.add(new Thread(h, "" + (i + 3)));  //Creamos e iniciamos 10 hilos
        hilos.get(i).start();
    }

    t1.start();t2.start();t3.start();t4.start(); //Se inicializan los hilos para comenzar su ejecucion

    t1.join();t2.join();t3.join();t4.join();//????
    for(int i=0; i<10; i++){
        hilos.get(i).join();    //"Cerramos" los 10 hilos que creamos
    }
}
```

- **Clase Hilos2:** Nuevamente, sólo hay variables de hilo pues todas viven en la definición de t1 o en la definición del hilo principal main.

```java
public void imprimeAlgo(){ //Metodazo que hara algo con estilo B)
    System.out.println("Imprimiendo con Estilo xD");
}

public static void main(String[] args) throws InterruptedException{
    Thread t1 = new Thread( //Creamos un hilo
        new Runnable() { //Usamos un new Runnable() para implementarlo directamente aqui
        Hilos2 h = new Hilos2(); //Creamos instancia de la clase

        @Override
        public void run() { //definimos manualmente el comportamiento del Hilo
            h.imprimeAlgo();
        }
    },
     "Hilo1");//Nombre que le damos al hilo.

    t1.start(); //Inicializamos el hilo
    t1.join();//?????
}
```

11. Escribe lo aprendido sobre esta practica, así como tus conclusiones. También comparte si tuviste dificultades y los descubrimientos o alguna cosa de interés.
    En general fue una práctica bastante sencilla, me sirvió para asentar los conocimientos sobre hilos y procesos que adquirí durante mi curso de sistemas operativos. Aunque apenas vimos una pequeña introducción al uso de hilos en java, podemos ver como esta herramienta podría ser empleada para mejorar el rendimiento de programas como los que hemos desarrollado a lo largo de la carrera. Así como sus ventajas, podemos también apreciar algunos de los desafíos que supone el uso compartido de recursos y que tan importante es una sincronización adecuada de los hilos, esto con el fin de obtener los resultados deseados de forma estable.
    Por último, añado aquí la captura del código solicitada, para la primera parte de la práctica donde se nos pide crear, inicializar y detener 10 hilos, aunque ya se visualiza en las capturas de la clase hilos/Hilos.

```java
ArrayList<Thread> hilos = new ArrayList<Thread>();       //Estructura para guardar 10 hilos

for(int i=0; i< 10; i++){
    hilos.add(new Thread(h, "" + (i + 3)));  //Creamos e iniciamos 10 hilos
    hilos.get(i).start();
}

t1.start();t2.start();t3.start();t4.start(); //Se inicializan los hilos para comenzar su ejecucion

t1.join();t2.join();t3.join();t4.join();//????
for(int i=0; i<10; i++){
    hilos.get(i).join();    //"Cerramos" los 10 hilos que creamos
}
```

Nota: No comenté todo el código de esta práctica pues la mayor parte del mismo no es de mi creación y Kass me dijo que sólo comentara lo que yo hice, por motivos similares, no se tomaron en consideración los errores señalados por sonarlint.

## Referencias


- Baeldung (8 de enero de 2024). The Thread.join() Method in Java. Baeldung https://www.baeldung.com/java-thread-join
- Fadatare Ramesh (s.f.). InterruptedException in Java with Example. Java guides https://www.javaguides.net/2019/07/interruptedexception-in-java-with-example.html
- Baeldung (8 de enero de 2024). Runnable vs. Callable in Java. Baeldung https://www.baeldung.com/java-runnable-callable
- Geeksforgeeks (s.f.). Synchronization in Java. Geeksforgeeks https://www.geeksforgeeks.org/synchronization-in-java/
- Baeldung (8 de enero de 2024). How to Handle InterruptedException in Java, Bealdung https://www.baeldung.com/java-interrupted-exception