Programación multi-thread en Java. Procesos asíncronos usando hilos

Cuando se crea una aplicación que puede ser utilizada por múltiples instancias (personas, máquinas y demás) simultáneamente (típico caso de un cliente - servidor) es muy común la creación de hilos para cada una de las instancias que ingresa o interactúa con nuestro sistema permitiendo así evitar problemas con lo que se conoce como *código bloqueante*. El código bloqueante, es un segmento de código (un método o función) que tarda una cantidad considerable de tiempo en ejecutarse (en tiempos de computación más de 5 segundos es muy significativo) y que de no ser por tener diferentes hilos asignados para cada usuario ejecutándose independientemente uno del otro, bloquearía la aplicación para los demás mientras alguno realiza el proceso bloqueante. Comentemos un corto ejemplo (cotidiano) para ilustrar esto.

Ejemplo para programación multi-hilos en Java

En ejemplo muy sencillo en el cual se puede evidenciar la utilidad de los hilos en Java es cuando se quieren mostrar mensajes por pantalla y que conlleven un proceso de tiempo significativo. Supongamos que queremos mostrar por pantalla los números del 1 al 100 añadiéndole los valores de algún atributo, esto es un proceso relativamente rápido que tomaría menos de un segundo pero esto quiere decir que si 100 usuarios quieren usar nuestro sistema, el usuario numero 100 deberá esperar 1 segundo del primero, 1 segundo del siguiente y así sucesivamente, tendría que esperar por lo menos unos 1 minuto (quizá un poco menos). Sin embargo si nuestro sistema se implementara usando programación multi-hilos (en java para nuestro caso) podríamos asignar un hilo para cada usuario y así podríamos estar mostrándole resultados a cada usuario sin tener que esperar a que el anterior termine su proceso.

Debo mencionar, que al tener múltiples hilos hay una problemática con los recursos compartidos (por ejemplo una impresora), y a la consistencia de la información, pues si múltiples usuarios (uno en cada hilo) quieren acceder a un mismo y único recurso de manera simultánea, ponen en peligro la estabilidad del sistema y la consistencia de la información. Esto se soluciona con métodos sincronizados pero hablaré de ellos luego en otra sección.

Código del ejemplo de hilos en Java

Vamos entonces a ver las dos versiones para nuestro ejemplo una sin hilos y otra con hilos. Debo aclarar desde ya que el tiempo que va a requerir cada ejecución (al menos para este ejemplo) sea con o sin hilos va a ser muy similar, pues al fin y al cabo se deben realizar las mismas tareas, lo que si cambia es el orden en que se realizan, cada hilo de irá alternando y realizando sus tareas, según se lo permita el sistema operativo. De lo anterior vemos que los hilos no necesariamente hacen nuestro sistema más rápido (repito, no necesariamente) sino que lo harán más fluido atendiendo a cada hilo por cortos instantes de tiempo.

Ejecución de código sin hilos

```
package hilos; public class SinHilos { String atributo; public SinHilos(int i) { atributo = "algo" +
i; }public static void main(String[] args) { for (int i = 0; i < 4; i++) { SinHilos sh
= new SinHilos(i); sh.run(); } } public void run() { for (int i = 0; i < 100; i++) {
System.out.println(i + ": " + atributo); } } }
```

El ejemplo del código anterior es simple, tenemos un sistema que imprime por pantalla los número del 0 al 4 agregándole el valor de un atributo del objeto, este proceso se repite 4 veces. Dado que la implementación es sin

hilos, entonces el sistema esperará a que finalice el primer ciclo para comenzar con el segundo y así. El resultado de la ejecución de este código es el siguiente:

```
run:
0: algo0
1: algo0
2: algo0
3: algo0
4: algo0
0: algo1
1: algo1
2: algo1
3: algo1
4: algo1
0: algo2
1: algo2
2: algo2
3: algo2
4: algo2
0: algo3
1: algo3
2: algo3
```

.

Ejecución de código con hilos java

La implementación de hilos se puede hacer de dos formas, usando la clase *Thread* o usando la interfaz *Runnable* e implementando el método *run()* (que, casualmente, así se lo llamamos en el caso sin hilos :P), luego para invocar cada hilo usamos el método *start()* y es básicamente todo, veamos:

Usando la clase *Thread*

```
package hilos; public class ConHilos extends Thread { String atributo; publicConHilos(int i) { atributo = "algo" + i; } public static void main(String[] args) { for (int i = 0; i < 4; i++) { ConHilos ch = new ConHilos(i); ch.start(); } } public void run() { for (int i = 0; i < 5; i++) { System.out.println(i + ": " + atributo); } } }
```

El código anterior extiende de *Thread*, implementa el método *run()* y dentro del ciclo for del main va creando los objetos y ejecutando su correspondiente hilos llamando a *start()*. En este caso dado que la ejecución es asíncrona, ya no esperará a que cada ciclo termine sino que irá atendiendo de una cada vez que sea posible. El resultado de esta ejecución es el siguiente:

```
run:
0: algo0
0: algo3
1: algo3
0: algo2
0: algo1
1: algo1
1: algo2
2: algo3
3: algo3
1: algo0
4: algo3
2: algo2
3: algo2
4: algo2
2: algo1
3: algo1
4: algo1
2: algo0
```

Usando la interfaz *Runnable*

```
package hilos; public class ConHilos implements Runnable { String atributo; public ConHilos(int i) { atributo = "algo" + i; } public static void main(String[] args) { for (int i = 0; i < 4; i++) {
```

```
Thread ch = newThread(new ConHilos(i)); ch.start(); } } public void run() { for (int i = 0; i < 5;
i++) { System.out.println(i + ": " + atributo); } } }
```

La diferencia fundamental es que en vez de extender de *Thread* extendemos de *Runnable* y al momento de crear los objetos lo hacemos usando el constructor de la clase *Thread* enviándole como argumento un objeto de la clase *ConHilos*que implementó a *Runnable*. El resultado de esta ejecución es simular al anterior con *Thread* como se aprecia en la figura. Nótese que el orden fue diferente debido a que es el sistema operativo quien decide de manera "aleatoria" cuando darle paso a cada hilo.

```
run:
0: algo0
1: algo0
0: algo2
0: algo1
1: algo1
1: algo2
2: algo2
0: algo3
2: algo0
1: algo3
3: algo2
2: algo1
4: algo2
2: algo3
3: algo3
3: algo0
4: algo3
3: algo1
```