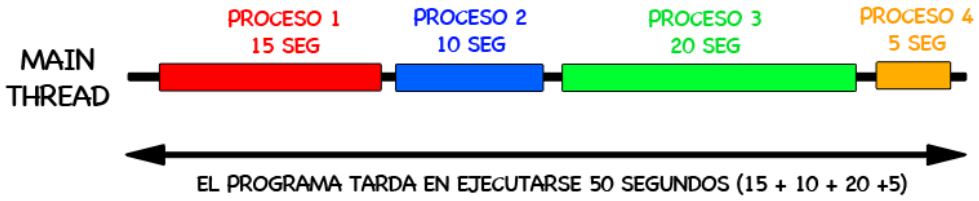
Multitarea e Hilos en Java con ejemplos (Thread & Runnable)

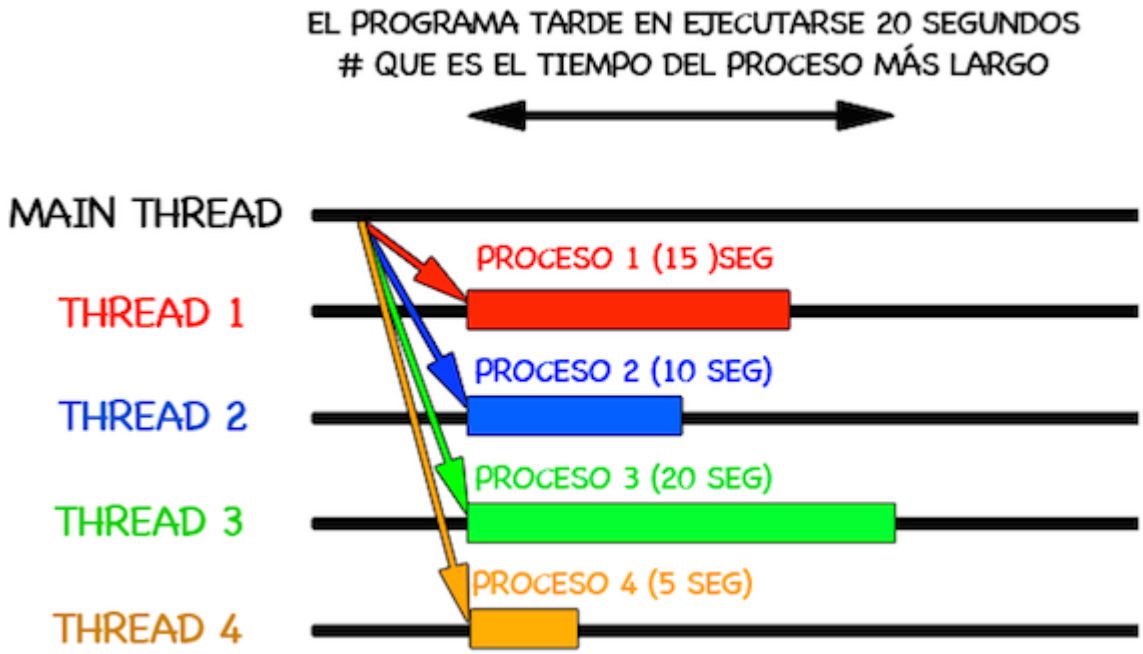
El proyecto de este post lo puedes descargar pulsando AQUI.

En esta entrada vamos a ver las diferentes maneras de cómo trabajar con Threads en Java (o hilos en español). Si no tienes muy claro el concepto de la multitarea te recomendamos que te leas primero la entrada de Multitarea e Hilos, fácil y muchas ventajas aunque en esta entrada también veremos (en menor detalle) los conceptos y las ventajas de la multitarea.

En esencia la multitarea nos permite ejecutar varios procesos a la vez; es decir, de forma concurrente y por tanto eso nos permite hacer programas que se ejecuten en menor tiempo y sean más eficientes. Evidentemente no podemos ejecutar infinitos procesos de forma concurrente ya que el hardware tiene sus limitaciones, pero raro es a día de hoy los ordenadores que no tengan más de un núcleo por tanto en un procesador con dos núcleos se podrían ejecutar dos procesos a la vez y así nuestro programa utilizaría al máximo los recursos hardware. Para que veáis la diferencia en un par de imágenes, supongamos que tenemos un programa secuencial en el que se han de ejecutar 4 procesos; uno detrás de otro, y estos tardan unos segundos:



Si en vez de hacerlo de forma secuencial, lo hiciésemos con 4 hilos, el programa tardaría en ejecutarse solo 20 segundos, es decir el tiempo que tardaría en ejecutarse el proceso más largo. Esto evidentemente sería lo ideal, pero la realidad es que no todo se puede paralelizar y hay que saber el número de procesos en paralelo que podemos lanzar de forma eficiente. En principio en esta entrada no vamos a hablar sobre ello ya que el objetivo de la misma es ver como se utilizan los hilos en java con un ejemplo relativamente sencillo y didáctico.



En Java para utilizar la multitarea debemos de usar la clase ***Thread*** (es decir que la clase que implementemos debe heredar de la clase Thread) y la clase Thread implementa la Interface ***Runnable***. En el siguiente diagrama de clase mostramos la Interface Runnable y la clase Thread con sus principales métodos:

En esta entrada no vamos a ver cómo utilizar todos los métodos de la clase Thread, pero os los mostramos para que sepáis que existen y a parte por su nombre podéis intuir su funcionalidad.

En esta entrada vamos a poner un ejemplo para que veáis las ventajas de la multitarea, viendo cómo se ejecutaría un programa sin utilizar la multitarea y otro utilizándola.

En este ejemplo vamos a simular el proceso de cobro de un supermercado; es decir, unos clientes van con un carro lleno de productos y una cajera les cobra los productos, pasándolos uno a uno por el escáner de la caja registradora. En este caso la cajera debe de procesar la compra cliente a cliente, es decir que primero le cobra al cliente 1, luego al cliente 2 y así sucesivamente. Para ello vamos a definir una clase "Cajera" y una clase "Cliente" el cual tendrá un "array de enteros" que representaran los productos que ha comprado y el tiempo que la cajera tardará en pasar el producto por el escáner; es decir, que si tenemos un array con [1,3,5] significará que el cliente ha comprado 3 productos y que la cajera tardara en procesar el producto 1 '1 segundo', el producto 2 '3 segundos' y el producto 3 en '5 segundos', con lo cual tardara en cobrar al cliente toda su compra '9 segundos'.

Explicado este ejemplo vamos a ver cómo hemos definido estas clases:

Clase "***Cajera.java***":

```
public class Cajera {

	private String nombre;

	// Constructor, getter y setter

	public void procesarCompra(Cliente cliente, long timeStamp)
{

		System.out.println("La cajera " + this.nombre +
				" COMIENZA A PROCESAR LA COMPRA DEL
CLIENTE " + cliente.getNombre() +
				" EN EL TIEMPO: " +
(System.currentTimeMillis() - timeStamp) / 1000    +
				"seg");

		for (int i = 0; i < cliente.getCarroCompra().length;
i++) {

	this.esperarXsegundos(cliente.getCarroCompra()[i]);
```

```
                                System.out.println("Procesado el
producto " + (i + 1) +
                                " ->Tiempo: " +
(System.currentTimeMillis() - timeStamp) / 1000 +
                                "seg");
                }

                System.out.println("La cajera " + this.nombre + " HA
TERMINADO DE PROCESAR " +
                                cliente.getNombre() + " EN EL TIEMPO:
" +
                                (System.currentTimeMillis() -
timeStamp) / 1000 + "seg");

        }


        private void esperarXsegundos(int segundos) {
                try {
                        Thread.sleep(segundos * 1000);
                } catch (InterruptedException ex) {
                        Thread.currentThread().interrupt();
                }
        }

}
```

Clase "***Cliente.java***":

```
public class Cliente {

        private String nombre;
        private int[] carroCompra;

        // Constructor, getter y setter

}
```

Si ejecutásemos este programa propuesto con dos Clientes y con un solo proceso (que es lo que se suele hacer normalmente), se procesaría primero la compra del Cliente 1 y después la del Cliente 2, con lo cual se tardará el tiempo del Cliente 1 + Cliente 2. A continuación vamos a ver como programamos el método Main para lanzar el programa. ***CUIDADO***: *Aunque hayamos puesto dos objetos de la clase Cajera (cajera1 y cajera2) no significa que tengamos dos cajeras independientes, lo que estamos diciendo es que dentro del mismo hilo se ejecute primero los métodos de la cajera1 y después los métodos de la cajera2, por tanto a nivel de procesamiento es como si tuviésemos una sola cajera*:

Clase "***Main.java***":

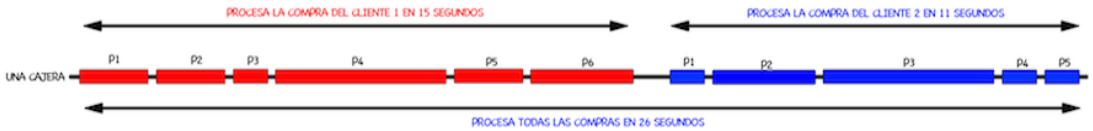
```
public class Main {

	public static void main(String[] args) {

		Cliente cliente1 = new Cliente("Cliente 1", new
int[] { 2, 2, 1, 5, 2, 3 });
		Cliente cliente2 = new Cliente("Cliente 2", new
int[] { 1, 3, 5, 1, 1 });

		Cajera cajera1 = new Cajera("Cajera 1");
		Cajera cajera2 = new Cajera("Cajera 2");

		// Tiempo inicial de referencia
		long initialTime = System.currentTimeMillis();

		cajera1.procesarCompra(cliente1, initialTime);
		cajera2.procesarCompra(cliente2, initialTime);
	}
}
```

Si ejecutamos este código tendremos lo siguiente:

```
La cajera Cajera 1 COMIENZA A PROCESAR LA COMPRA DEL
CLIENTE Cliente 1 EN EL TIEMPO: 0seg

Procesado el producto 1 ->Tiempo: 2seg

Procesado el producto 2 ->Tiempo: 4seg

Procesado el producto 3 ->Tiempo: 5seg

Procesado el producto 4 ->Tiempo: 10seg

Procesado el producto 5 ->Tiempo: 12seg

Procesado el producto 6 ->Tiempo: 15seg

La cajera Cajera 1 HA TERMINADO DE PROCESAR Cliente 1
EN EL TIEMPO: 15seg

La cajera Cajera 2 COMIENZA A PROCESAR LA COMPRA DEL
CLIENTE Cliente 2 EN EL TIEMPO: 15seg

Procesado el producto 1 ->Tiempo: 16seg

Procesado el producto 2 ->Tiempo: 19seg

Procesado el producto 3 ->Tiempo: 24seg
```

```
Procesado el producto 4 ->Tiempo: 25seg

Procesado el producto 5 ->Tiempo: 26seg

La cajera Cajera 2 HA TERMINADO DE PROCESAR Cliente 2
EN EL TIEMPO: 26seg
```

Como vemos se procesa primero la compra del cliente 1 y después la compra del cliente 2 tardando en procesar ambas compras un tiempo de 26 segundos.



¿Y si en vez de procesar primero un cliente y después otro, procesásemos los dos a la vez?, ¿Cuanto tardaría el programa en ejecutarse?. Pues bien si en vez de haber solo una Cajera (es decir un solo hilo), hubiese dos Cajeras (es decir dos hilos o threads) podríamos procesar los dos clientes a la vez y tardar menos tiempo en ejecutarse el programa. Para ello debemos de modificar la clase "***Cajera.java***" y hacer que esta clase herede de la clase ***Thread*** para heredar y sobre-escribir algunos de sus métodos. Primero vamos a ver como codificamos esta nueva clase "***CajeraThread.java***" y después explicamos sus caracteristicas.

```java
public class CajeraThread extends Thread {

	private String nombre;

	private Cliente cliente;

	private long initialTime;

	// Constructor, getter & setter

	@Override
	public void run() {

		System.out.println("La cajera " + this.nombre + "
COMIENZA A PROCESAR LA COMPRA DEL CLIENTE "
					+ this.cliente.getNombre() + "
EN EL TIEMPO: "
					+ (System.currentTimeMillis() -
this.initialTime) / 1000
					+ "seg");

		for (int i = 0; i <
this.cliente.getCarroCompra().length; i++) {
```

```
            this.esperarXsegundos(cliente.getCarroCompra()[i]);
                        System.out.println("Procesado el producto " +
(i + 1)
                        + " del cliente " + this.cliente.getNombre()
+ "->Tiempo: "
                        + (System.currentTimeMillis() -
this.initialTime) / 1000
                        + "seg");
                }

                System.out.println("La cajera " + this.nombre + " HA
TERMINADO DE PROCESAR "
                                                        +
this.cliente.getNombre() + " EN EL TIEMPO: "
                                                        +
(System.currentTimeMillis() - this.initialTime) / 1000
                                                        + "seg");
        }

        private void esperarXsegundos(int segundos) {
                try {
                        Thread.sleep(segundos * 1000);
                } catch (InterruptedException ex) {
                        Thread.currentThread().interrupt();
                }
        }

}
```

Lo primero que vemos y que ya hemos comentado es que la clase "CajeraThread" debe de heredar de la clase Thread: "***extendsThread***".

Otra cosa importante que vemos es que hemos sobre-escrito el método "***run()***" (de ahí la etiqueta *@Override*) . Este método es imprescindibles sobre-escribirlo (ya que es un método que está en la clase Runnable y la clase Thread Implementa esa Interface) porque en él se va a codificar la funcionalidad que se ha de ejecutar en un hilo; es decir, que lo que se programe en el método "***run()***" se va a ejecutar de forma secuencial en un hilo. En esta clase "CajeraThread" se pueden sobre-escribir más métodos para que hagan acciones sobre el hilo o thread como por ejemplo, parar el thread, ponerlo en reposos, etc. A continuación vamos a ver como programamos el método Main para que procese a los clientes de forma paralela y ver como se tarda menos en procesar todo. El método Main está en la clase "***MainThread.java***" que tiene el siguiente contenido:

```
public class MainThread {

        public static void main(String[] args) {
```

```
                Cliente cliente1 = new Cliente("Cliente 1", new
int[] { 2, 2, 1, 5, 2, 3 });
                Cliente cliente2 = new Cliente("Cliente 2", new
int[] { 1, 3, 5, 1, 1 });

                // Tiempo inicial de referencia
                long initialTime = System.currentTimeMillis();
                CajeraThread cajera1 = new CajeraThread("Cajera 1",
cliente1, initialTime);
                CajeraThread cajera2 = new CajeraThread("Cajera 2",
cliente2, initialTime);

                cajera1.start();
                cajera2.start();
        }
}
```

Ahora vamos a ver cuál sería el resultado de esta ejecución y vamos a comprobar como efectivamente el programa se ejecuta de forma paralela y tarda solo 15 segundos en terminar su ejecución:

```
La cajera Cajera 1 COMIENZA A PROCESAR LA COMPRA DEL
CLIENTE Cliente 1 EN EL TIEMPO: 0seg

La cajera Cajera 2 COMIENZA A PROCESAR LA COMPRA DEL
CLIENTE Cliente 2 EN EL TIEMPO: 0seg

Procesado el producto 1 del cliente Cliente 2->Tiempo:
1seg

Procesado el producto 1 del cliente Cliente 1->Tiempo:
2seg

Procesado el producto 2 del cliente Cliente 2->Tiempo:
4seg

Procesado el producto 2 del cliente Cliente 1->Tiempo:
4seg

Procesado el producto 3 del cliente Cliente 1->Tiempo:
5seg

Procesado el producto 3 del cliente Cliente 2->Tiempo:
9seg

Procesado el producto 4 del cliente Cliente 2->Tiempo:
10seg
```

```
Procesado el producto 4 del cliente Cliente 1->Tiempo:
10seg

Procesado el producto 5 del cliente Cliente 2->Tiempo:
11seg

La cajera Cajera 2 HA TERMINADO DE PROCESAR Cliente 2
EN EL TIEMPO: 11seg

Procesado el producto 5 del cliente Cliente 1->Tiempo:
12seg

Procesado el producto 6 del cliente Cliente 1->Tiempo:
15seg

La cajera Cajera 1 HA TERMINADO DE PROCESAR Cliente 1
EN EL TIEMPO: 15seg
```

En este ejemplo vemos como el efecto es como si dos cajeras procesasen la compra de los clientes de forma paralela sin que el resultado de la aplicación sufra ninguna variación en su resultado final, que es el de procesar todas las compras de los clientes de forma independiente. De forma gráfica vemos que el programa ha realizado lo siguiente en dos hilos distintos:

Otra forma de hacer lo mismo pero sin heredar de la clase "Thread" es implementar la Interface "Runnable". En este caso no dispondremos ni podremos sobre-escribir los métodos de la clase Thread ya que no la vamos a utilizar y solo vamos a tener que sobre-escribir el método "***run()***". En este caso solo será necesario implementar el método "***run()***" para que los procesos implementados en ese método se ejecuten en un hilo diferente. Vamos a ver un ejemplo de como utilizando objetos de las clases "***Cliente.java***" y "***Cajera.java***" podemos implementar la multitarea en la misma clase donde se llama al método Main de la aplicación. A continuación vemos la codificación en la clase "***MainRunnable.java***":

```
public class MainRunnable implements Runnable{
```

```
        private Cliente cliente;
        private Cajera cajera;
        private long initialTime;

        public MainRunnable (Cliente cliente, Cajera cajera, long
initialTime){
                this.cajera = cajera;
                this.cliente = cliente;
                this.initialTime = initialTime;
        }

        public static void main(String[] args) {

                Cliente cliente1 = new Cliente("Cliente 1", new
int[] { 2, 2, 1, 5, 2, 3 });
                Cliente cliente2 = new Cliente("Cliente 2", new
int[] { 1, 3, 5, 1, 1 });

                Cajera cajera1 = new Cajera("Cajera 1");
                Cajera cajera2 = new Cajera("Cajera 2");

                // Tiempo inicial de referencia
                long initialTime = System.currentTimeMillis();

                Runnable proceso1 = new MainRunnable(cliente1,
cajera1, initialTime);
                Runnable proceso2 = new MainRunnable(cliente2,
cajera2, initialTime);

                new Thread(proceso1).start();
                new Thread(proceso2).start();

        }

        @Override
        public void run() {
                this.cajera.procesarCompra(this.cliente,
this.initialTime);
        }

}
```

En este caso implementamos el método "***run()***" dentro de la misma clase donde se encuentra el método Main, y en el llamamos al método de "procesarCompra()" de la clase Cajera. Dentro del método Main, nos creamos dos objetos de la misma clase en la que estamos ("new MainRunnable") y nos creamos dos objetos de la clase Thread para lanzar los proceso y que se ejecuten estos en paralelo. El resultado de esta ejecución es el mismo que en el caso anterior:

```
La cajera Cajera 2 COMIENZA A PROCESAR LA COMPRA DEL
CLIENTE Cliente 2 EN EL TIEMPO: 0seg

La cajera Cajera 1 COMIENZA A PROCESAR LA COMPRA DEL
CLIENTE Cliente 1 EN EL TIEMPO: 0seg

Procesado el producto 1 del cliente Cliente 2->Tiempo:
1seg

Procesado el producto 1 del cliente Cliente 1->Tiempo:
2seg

Procesado el producto 2 del cliente Cliente 2->Tiempo:
4seg

Procesado el producto 2 del cliente Cliente 1->Tiempo:
4seg

Procesado el producto 3 del cliente Cliente 1->Tiempo:
5seg

Procesado el producto 3 del cliente Cliente 2->Tiempo:
9seg

Procesado el producto 4 del cliente Cliente 2->Tiempo:
10seg

Procesado el producto 4 del cliente Cliente 1->Tiempo:
10seg

Procesado el producto 5 del cliente Cliente 2->Tiempo:
11seg

La cajera Cajera 2 HA TERMINADO DE PROCESAR Cliente 2
EN EL TIEMPO: 11seg

Procesado el producto 5 del cliente Cliente 1->Tiempo:
12seg

Procesado el producto 6 del cliente Cliente 1->Tiempo:
15seg

La cajera Cajera 1 HA TERMINADO DE PROCESAR Cliente 1
EN EL TIEMPO: 15seg
```

CONCLUSIONES Y ACLARACIONES:

El concepto de multitarea o multiprocesamiento es bastante sencillo de entender ya que solo consiste en hacer varias cosas a la vez sin que se vea alterado el resultado final.

Como ya se ha dicho en la entrada no todo se puede paralelizar y en muchas ocasiones suele ser complicado encontrar la manera de paralelizar procesos dentro de una aplicación sin que esta afecte al resultado de la misma, por tanto aunque el concepto sea fácil de entender el aplicarlo a un caso práctico puede ser complicado para que el resultado de la aplicación no se vea afectado.

Por otro lado para los que empecéis a ver estos temas de la concurrencia, multitarea y demás, no so preocupéis al principio si os cuesta programar problemas de este tipo ya que a parte de la multitarea se mezclan cosas como la herencia y las Interfaces que al principio son cosas que cuestan de asimilar, así que ir poco a poco pero tener muy claro que la multitarea es muy util y se ha de aplicar para hacer las aplicaciones más eficientes y que den mejor rendimiento.