Java Executor Service pertenece al API de Java7 y es una de las clases que nos permite gestionar la programación concurrente de una forma más sencilla y optima. Vamos a ver un ejemplo, para ello nos vamos a construir una clase Tarea que realice un pequeño bucle por pantalla

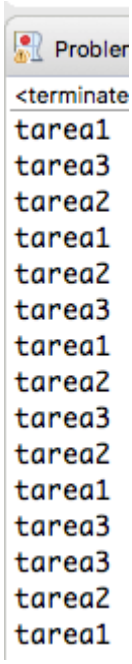
```
package com.arquitecturajava.executors;

public class Tarea implements Runnable {

	private String nombre;

	public Tarea(String nombre) {
		super();
		this.nombre = nombre;
	}

	@Override
	public void run() {

		for (int i = 0; i < 5; i++) {
			System.out.println(nombre);
			try {
				Thread.sleep(1000);
			} catch (InterruptedException e) {
				// TODO Auto-generated catch block
				e.printStackTrace();
			}
		}

	}
```

```
}
```

Como podemos ver es una clase normal que implementa el interface Runnable y que tiene un pequeño bucle que ejecutamos cada segundo. Vamos a crear un programa Main con 3 tareas y ejecutarlas con 3 hilos.

```
package com.arquitecturajava.executors;

public class Principal {

	public static void main(String[] args) {
		Thread t1= new Thread(new Tarea("tarea1"));
		t1.start();
		Thread t2= new Thread(new Tarea("tarea2"));
		t2.start();
		Thread t3= new Thread(new Tarea("tarea3"));
		t3.start();

	}

}
```

El resultado lo veremos salir por la consola:



La funcionalidad es un poco repetitiva .¿Podemos realizar de una forma mejor?

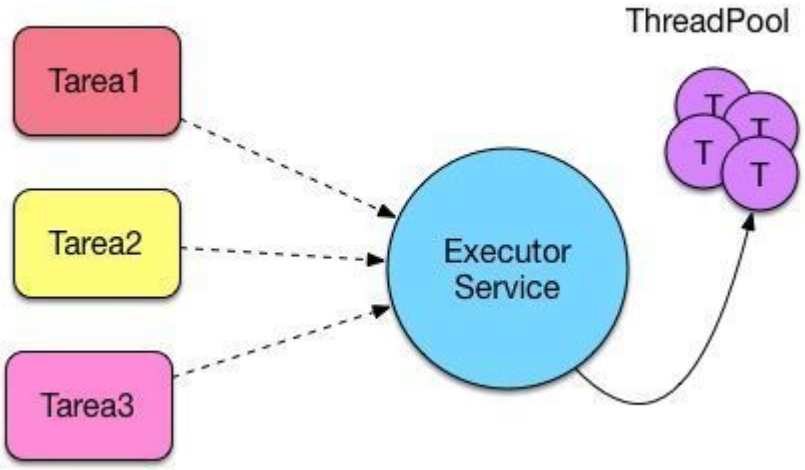
Java Executor Service y Streams

Vamos a construir un Stream con tres cadenas y convertir estas cadenas a objetos Runnables . Hecho esto invocaremos al método execute de un Executor Service que se encarga de automatizar la ejecución de cualquier objeto Runnable.

```
package com.arquitecturajava.executors;

import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.stream.Stream;

public class PrincipalExecutor {
```

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

                Stream<String> flujo = Stream.of("tarea1", "tarea2",
"tarea3");

                ExecutorService servicio =
Executors.newCachedThreadPool();
                flujo.map(t->new Tarea(t)).forEach(servicio::execute);
        }

}
```

El resultado será identico pero tendrá varias ventajas. En primer lugar la reducción de código y la flexibilidad que tendríamos si utilizáramos un stream infinito.En segundo lugar al no ser nosotros los que inicializamos los Threads .Dejamos al API de Java que se encargue de hacerlo de la forma más correcta. Por ejemplo en este caso si tuviéramos muchas tareas que ejetutar el API cachearía un numero determinado de Threads y usaría solo estos.



La clase Executor service es muy útil cuando nos encontramos en situaciones de

programación concurrente

Otros artículos relacionados : Java Streams , Java Lambda Java Threads