¿Para qué sirve un Java Callable interface?. Este interface esta ligado de forma importante a la programación concurrente. Cuando uno empieza trabajar en Java , rápidamente aparece la clase Thread que nos permite ejecutar tareas concurrentes .Sin embargo tiene algunas limitaciones,vamos a ver un ejemplo:

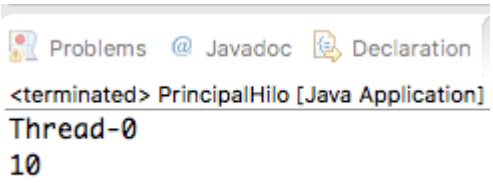
```
package com.arquitecturajava;

public class Tarea implements Runnable {

	@Override
	public void run() {
		int total = 0;
		for(int i=0;i<5;i++) {
			total+=i;
			try {
				Thread.sleep(300);
			} catch (InterruptedException e) {
				// TODO Auto-generated catch block
				e.printStackTrace();
			}
		}
		System.out.println(Thread.currentThread().getName());
		System.out.println(total);
	}

}
```

Acabamos de crear una tarea que implementa el interface Runnable. Podemos a partir de

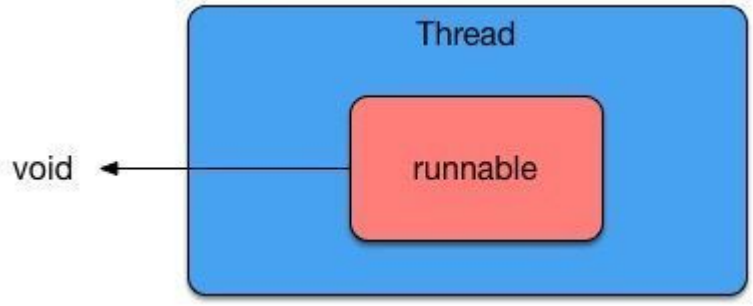
ella crear un Thread o hilo que la ejecute y nos imprima por pantalla la suma de los primeros 5 términos después de 1500 milisegundos (el bucle itera 5 veces) .

```
package com.arquitecturajava;

public class PrincipalHilo {

	public static void main(String[] args) {
		Tarea t= new Tarea();
		Thread hilo= new Thread(t);
		hilo.start();

	}

}
```

El resultado lo vemos aparecer por la consola:



Todo ha funcionado correctamente. El problema es que nos vemos obligados a imprimir los datos por la consola. El método run del interface Runnable no devuelve nada.



Java Callable Interface

En la mayor parte de las ocasiones necesitamos que se ejecute una tarea en paralelo y luego en el futuro nos devuelva un resultado. ¿Cómo podemos hacer esto? . Java provee para estas situaciones del interface Callable, vamos a verlo.

```
package com.arquitecturajava;

import java.util.concurrent.Callable;

public class MiCallable implements Callable<Integer> {

	@Override
	public Integer call() throws Exception {
		int total = 0;
		for(int i=0;i&lt;5;i++) {
			total+=i;
			try {
				Thread.sleep(300);
			} catch (InterruptedException e) {
```

```
                                // TODO Auto-generated catch block
                                e.printStackTrace();
                        }
                }
                System.out.println(Thread.currentThread().getName());
                return total;
        }

}
```

En este caso nos encontramos con algo muy similar pero usamos el interface Callable. Este interface dispone del método call que es capaz de devolvernos un resultado algo que el método run no permite.

public void run();

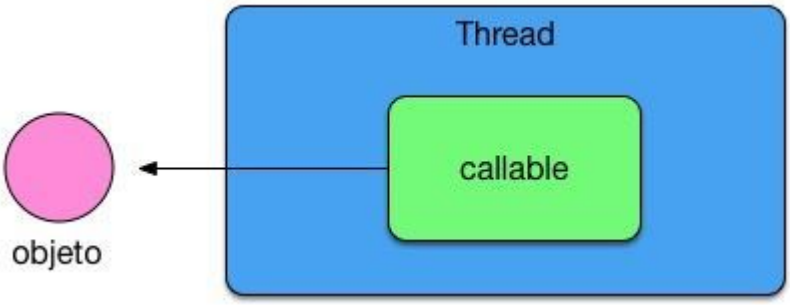
public T call();

Acabamos de crear una clase que implemente Java Callable interface. Es momento de usarla desde un método main.

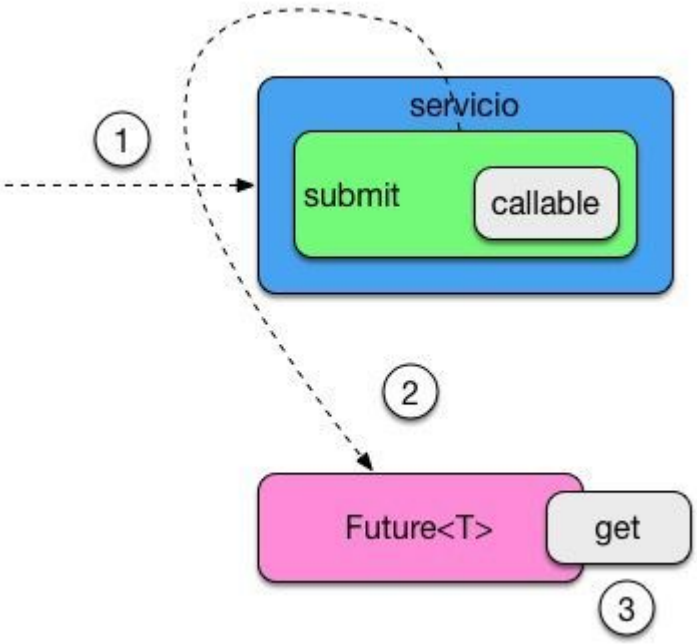
```
package com.arquitecturajava;

import java.util.concurrent.ExecutionException;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;
```

```
public class PrincipalCallable {

	public static void main(String[] args) {
		try {
			ExecutorService servicio=
Executors.newFixedThreadPool(1);

			Future<Integer> resultado= servicio.submit(new
MiCallable());
			if(resultado.isDone()) {
			System.out.println(resultado.get());
			}
		} catch (InterruptedException e) {
			// TODO Auto-generated catch block
			e.printStackTrace();
		} catch (ExecutionException e) {
			// TODO Auto-generated catch block
			e.printStackTrace();
		}

	}

}
```

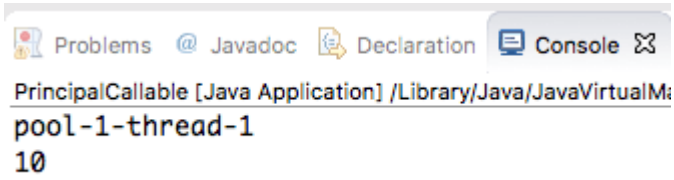
En este caso hemos usado ExecutorService para crear un pool de Thread con un único hilo y enviar la tarea al pool utilizando el método submit.



Cuando invoquemos el servicio recibiremos de forma automática una variable de tipo Future la cual recibirá en un futuro valor al cual 10 que podremos imprimir usando el método get().



El resultado en la consola es parecido , la única diferencia es que usamos un pool de Threads.



Hemos recibido datos de retorno de una tarea que se ejecuta en paralelo

Otros artículos relacionados : Java Executor Service y Threading ,Java wait notify y threads , JavaScript Promise y la programación asíncrona