Sumario

Introducción a Threads

Qué es un Thread (Hilo de Ejecución)

Un concepto fundamental en programación es la idea de manejar más de una tarea a la vez. Muchos problemas de programación requieren que el programa puede detener lo que esté haciendo, tratar con algún otro problema y regresar al proceso principal.

Dentro de un programa, las partes que corren separadamente se llaman hilos (Threads) y el concepto general se llama Multithreading.

Ordinariamente, los hilos son una manera de asignar el tiempo de un solo procesador. Pero si el sistema operativo apoya procesadores múltiples, cada hilo puede asignarse a un procesador diferente y ellos pueden correr realmente en paralelo. Una de las características convenientes de Multithreading es que el programador no necesita preocupar sobre si existe uno o más procesadores. El programa es lógicamente dividido en hilos y si la máquina tiene más de un procesador entonces el programa corre más rápidamente, sin necesidad de ajuste especial.

Qué es un proceso

Un proceso es un programa ejecutándose de forma independiente y con un espacio propio de memoria. Un Sistema Operativo multitarea es capaz de ejecutar más de un proceso simultáneamente. Un thread o hilo es un flujo secuencial simple dentro de un proceso. Un único proceso puede tener varios hilos ejecutándose. Por ejemplo, el programa Mozilla sería un proceso, mientras que cada una de las ventanas que se pueden tener abiertas simultáneamente trayendo páginas HTML estaría formada por al menos un hilo.

Un sistema multitarea da realmente la impresión de estar haciendo varias cosas a la vez y eso es una gran ventaja para el usuario. Sin el uso de threads hay tareas que son prácticamente imposibles de ejecutar, particularmente las que tienen tiempos de espera importantes entre etapas.

Los threads o hilos de ejecución permiten organizar los recursos de la PC de forma que pueda haber varios programas actuando en paralelo. Un hilo de ejecución puede realizar cualquier tarea que pueda realizar un programa normal y corriente. Bastará con indicar lo que tiene que hacer en el método run(), que es el que define la actividad principal de las threads.

Qué significa Multi Threading

La Máquina Virtual Java (JVM) es un sistema multi-thread. Es decir, es capaz de ejecutar varias secuencias de ejecución (programas) simultáneamente.

La programación multithreading (multi-hilo) permite la ocurrencia simultánea de varios flujos de control. Cada uno de ellos puede programarse independientemente y realizar un trabajo, distinto, idéntico o complementario, a otros flujos paralelos.

Un procesador ejecuta las instrucciones secuencialmente una después de la otra. A esta ejecución secuencial de instrucciones se le llama un Execution Thread (Hilo de Ejecución).

Multithreading es cuando se están ejecutando varios threads simultáneamente Si un procesador ejecuta las instrucciones secuencialmente, una después de la otra ¿Cómo le hace para ejecutar

varios threads simultáneamente? No puede hacerlo, pero puede simular que lo hace.

A continuación se detalla la arquitectura de trabajo de multithreading utilizando un CPU únicamente, como se da en la mayoría de los casos:

Creación de Threads

Alternativas de creación

En Java hay dos formas de crear nuevos threads. La primera de ellas consiste en crear una nueva clase que herede de la clase java.lang.Thread y sobrecargar el método **run()** de dicha clase. El segundo método consiste en declarar una clase que implemente la interface java.lang.Runnable, la cual declarará el método run(); posteriormente se enviará dicho objeto a la clase Thread.

En este caso, se ha creado la clase MiClase, que hereda de Thread. La principal actividad del thread, para que escriba 10 veces el nombre del thread creado.

Para poner en marcha este nuevo thread se debe llamar al método **start()**, heredado de la superClase Thread, que se encarga de llamar a **run()**.

Creación a través de la clase Thread

Veamos un ejemplo creando una clase que hereda de la clase java.lang.Thread:

```
public class MiClase extends Thread{
	private String descripcion;
	public MiClase(String string) {
		descripcion=string;
	}
	@Override
	public void run() {
		for(int i=0;i<10;i++)
		   System.out.println(descripcion+": Este es el thread "+getName());
	}
}
```

Para realizar la ejecución de varios threads de la misma clase los instanciamos y luego llamamos al método start():

```
public class Main {
	public static void main(String[] args) {
		MiClase m1 = new MiClase("m1");
```

```
            //m1.run();       //El método run() no se ejecuta en un nuevo Hilo
            m1.start();       //El método start() invoca el método run() en un
                              //nuevo Hilo
            MiClase m2 = new MiClase("m2");
            m2.start();

            new MiClase("m3").start();    //invocamos un nuevo hilo sin
                                          //referencia de variable.

            //Obteniendo hilos en ejecución para recuperar la referencia
            //Se recorre el mapa de hilos del proceso actual.
            Thread
                  .getAllStackTraces()
                  .forEach((k,v)->System.out.println(k+" "+v));

            Thread
                  .getAllStackTraces().forEach((k,v)→
                  .forEach((k,v)-> System.out.println(k.getName()));

      }
}
```

Creación a través de la interfaz Runnable

A continuación se presenta el mismo ejemplo, pero esta vez implementando la interfaz Runnable en lugar de heredar de la clase Thread:

```
public class MiClase implements Runnable{
      private String descripcion;
      public MiClase(String string) {
            descripcion=string;
      }
      public void run() {
            for(int i=0;i<10;i++)
            System.out.println("Este es el thread "+descripcion);
      }
}
```

Para realizar la ejecución de varios threads de la misma clase los instanciamos y luego llamamos al método **start()**:

```
public class Main {
      public static void main(String[] args) {
            MiClase m1 = new MiClase("m1");
            Thread thread = new Thread(m1);
            thread.start();

            MiClase m2 = new MiClase("m2");
            Thread thread2 = new Thread(m2);
            thread2.start();

            //Creación de un Thread sin referencia de variable.
            new Thread(new MiClase("m3").start();
      }
}
```

El propósito de usar la interface Runable es no usar la herencia en la clase que inicia un Thread, debido a que muchas veces la herencia ya esta ocupada, y en java una clase no puede heredad de mas de una superclase.

Manejo de Threads

Los métodos start() y run()

Como habíamos mencionado antes, para poner en marcha este nuevo thread se debe llamar al método **start()**, heredado de la super-clase **Thread**, que se encarga de llamar a **run()**. Es importante no confundir el método **start** con el método **run**. El método **run** contiene el código a ser ejecutado "asíncronamente" en otro thread, mientras que el método **start** es el que crea el thread y en algún punto hace que ese thread ejecute lo que está en **run**. Este método devuelve el control inmediatamente. Pero si mezclamos todo y ejecutamos directamente **run()**, el código se

ejecutará en el thread actual!

El método **start()** devuelve el control inmediatamente... mientras tanto, el nuevo thread inicia su recorrido por el método **run()**. ¿Hasta cuándo? Hasta que termina ese método, cuando sale, termina el thread.

El método join()

Si un thread necesita esperar a que otro termine (por ejemplo el thread padre esperar a que termine el hijo) puede usar el método join(). ¿Por qué se llama así? Crear un proceso es como una bifurcación, se abren dos caminos, que uno espere a otro es lo contrario, una unificación.

A continuación se presenta un ejemplo más complejo: una reunión de alumnos. El siguiente ejemplo usa threads para activar simultáneamente tres objetos de la misma clase, que comparten los recursos del procesador peleándose para escribir a la pantalla.

```
public static void main(String args[]) throws InterruptedException {
      Thread juan = new Thread (new Alumno("Juan"));
      Thread pepe = new Thread (new Alumno ("Pepe"));
      Thread andres = new Thread (new Alumno ("Andres"));
      juan.start();
      juan.join();
      pepe.start();
      pepe.join();
      andres.start();
      andres.join();
}
```

El método join() qué llamamos al final hace que el programa principal espere hasta que este thread esté "muerto" (finalizada su ejecución). Este método puede disparar la excepción InterruptedException, por lo que lo hemos tenido en cuenta en el encabezamiento del método.

En nuestro ejemplo, simplemente a cada instancia de Amigo(...) que creamos la hemos ligado a un thread y puesto a andar. Corren todas en paralelo hasta que mueren de muerte natural, y también el programa principal acaba.

```
class Alumno implements Runnable {
	String mensaje;
	public Alumno(String nombre) {
		mensaje = "Hola, soy "+nombre+" y este es mi mensaje ";
	}
	@Override
	public void run() {
		for (int i=1; i<6; i++) {
			String msg = mensaje+i;
			System.out.println(msg);
		}
	}
}
```

En un sistema operativo preemptivo, la salida será más o menos así:

```
Hola, soy Juan y este es mi mensaje 1
Hola, soy Juan y este es mi mensaje 2
Hola, soy Luis y este es mi mensaje 1
Hola, soy Luis y este es mi mensaje 2
Hola, soy Nora y este es mi mensaje 1
Hola, soy Nora y este es mi mensaje 2
Hola, soy Nora y este es mi mensaje 3
Hola, soy Juan y este es mi mensaje 3
...........etc.
```

Qué significa que un sistema operativo es preemptivo? Casos típicos son Unix o Windows: cada tarea utiliza una parte del tiempo del procesador, y luego lo libera para que puedan ejecutarse otras tareas (otros threads). Por eso se mezclan los mensajes de salida. Si el sistema operativo es no preemptivo, el procesador no se libera hasta que no termina con el thread actual, y por lo tanto la salida sería así:

```
Hola, soy Juan y este es mi mensaje 1
Hola, soy Juan y este es mi mensaje 2
Hola, soy Juan y este es mi mensaje 3
Hola, soy Juan y este es mi mensaje 4
Hola, soy Juan y este es mi mensaje 5
...........etc.
```

El método yield()

Si se está utilizando un sistema operativo no preemptivo, deben explícitamente indicarle al procesador cuando puede ejecutar (dar paso) a otra tarea; para eso simplemente modifique el método **run()**:

```
@Override
public void run() {
```

```
      for (int i=1; i<6; i++) {
      String msg = mensaje+i;
      System.out.println(msg);
      Thread.yield();
      }
}
```

En este ejemplo, tanto en sistemas preemptivos como no preemptivos la salida será:

```
Hola, soy Juan y este es mi mensaje 1
Hola, soy Luis y este es mi mensaje 1
Hola, soy Nora y este es mi mensaje 1
Hola, soy Juan y este es mi mensaje 2
Hola, soy Luis y este es mi mensaje 2
Hola, soy Nora y este es mi mensaje 2
Hola, soy Juan y este es mi mensaje 3
Hola, soy Luis y este es mi mensaje 3
...........etc.
```

Esto es porque enseguida de imprimir estamos liberando al procesador para que pase a otro thread (si hay alguno esperando). Noten la diferencia con el primer caso, sin usar yield(), para sistemas preemptivos: el procesador reparte su trabajo en forma (aparentemente) impredecible, por eso el orden de los mensajes no será el mismo en cualquier máquina o sistema operativo.

El método sleep()

El método **sleep()** simplemente le dice al thread que duerma durante los milisegundos especificados. Se debería utilizar **sleep()** cuando se pretenda retrasar la ejecución del thread, **sleep()** no consume recursos del sistema mientras el thread duerme. De esta forma otros threads pueden seguir funcionando.

```
objThread.sleep(100);
```

El método suspend()

Puede resultar útil suspender la ejecución de un thread sin marcar un límite de tiempo. Si, por ejemplo, está construyendo un applet con un thread de animación, querrá permitir al usuario la opción de detener la animación hasta que quiera continuar. No se trata de terminar la animación, sino desactivarla. Para este tipo de control de thread se puede utilizar el método **suspend()**.

```
objThread.suspend();
```

Este método se encuentra deprecado, no es recomendable su utilización porque puede generar situaciones de deadlock.

El método resume()

Cuando el thread es suspendido indefinidamente (con el método suspend()) se puede volver a activarlo de nuevo con el método resume():

```
objThread.resume();
```

Este método se encuentra deprecado, no es recomendable su utilización porque puede generar situaciones de deadlock.

El método getName()

Este método devuelve el valor actual, de tipo cadena, asignado como nombre al hilo en ejecución mediante setName().

El método setName(String)

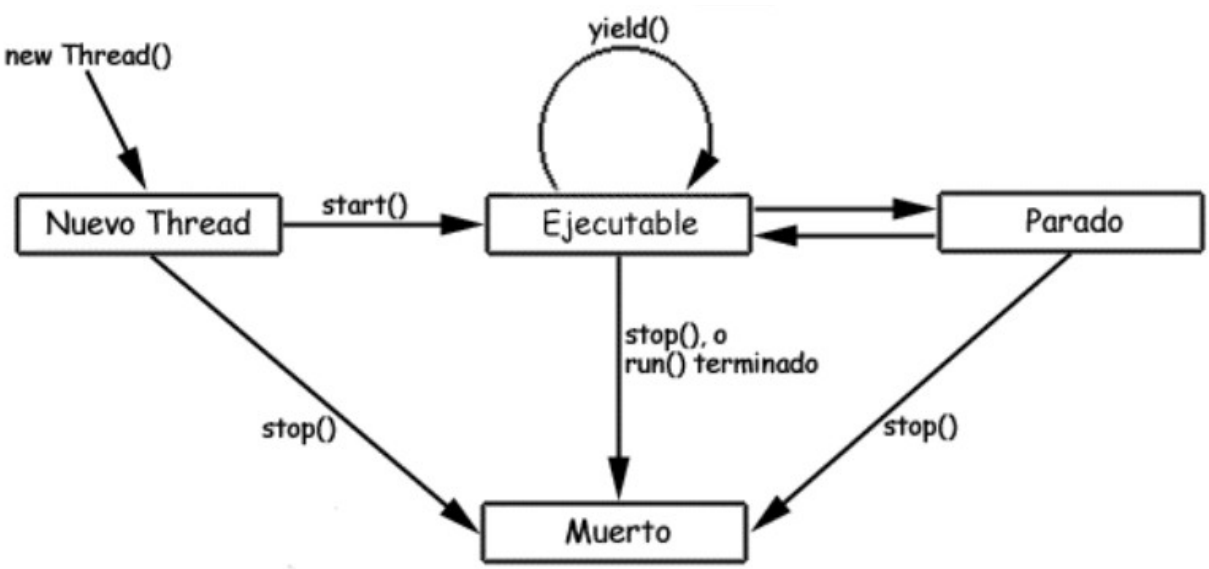
Este método permite identificar al hilo con un nombre menmónico. De esta manera se facilita la depuración de programas multihilo. El nombre mnemónico aparecerá en todas las líneas de trazado que se muestran cada vez que el intérprete Java imprime excepciones no capturadas.

Ciclo de vida

Qué es el ciclo de vida

El ciclo de vida de un thread representa los estados por cuales puede pasar un thread desde que nace hasta que muere. Durante el ciclo de vida de un thread, éste se puede encontrar en diferentes estados. La figura siguiente muestra estos estados y los métodos que provocan el paso de un estado a otro.

Diagrama de ciclo de vida



Estados de un Thread

Estado Nuevo

El thread ha sido creado pero no inicializado, es decir, no se ha ejecutado todavía el método **start()**. Se producirá un mensaje de error (IllegalThreadStateException) si se intenta ejecutar cualquier método de la clase Thread distinto de start().

Estado Ejecutable

El thread puede estar ejecutándose, siempre y cuando se le haya asignado un determinado tiempo de CPU. En la práctica puede no estar siendo ejecutado en un instante determinado en beneficio de otro thread.

Estado Bloqueado

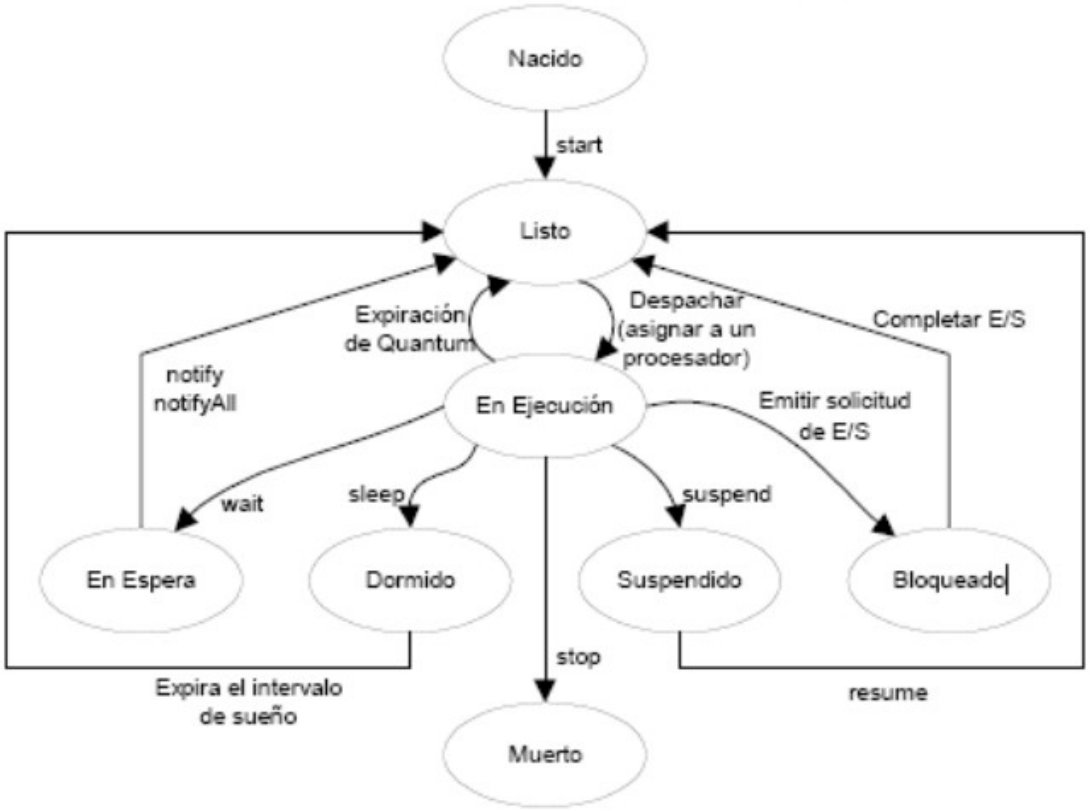
El thread podría estar ejecutándose, pero hay alguna actividad interna suya que lo impide, como por ejemplo una espera producida por una operación de escritura o lectura de datos por teclado (E/S). Si un thread está en este estado, no se le asigna tiempo de CPU.

Estado Muerto

La forma habitual de que un thread muera es finalizando el método **run()**. También puede llamarse al método **stop()** de la clase Thread, aunque dicho método es considerado “peligroso” y no se debe utilizar.

Ampliación del ciclo de vida

Ampliando los conceptos del diagrama anterior, un Thread desde su ejecución hasta su muerte pasa por diversos estados de vida. Estos estados y sus transiciones se muestran en forma más detallada en el diagrama siguiente:



Un thread tras su creación pasa a estado LISTO de ahí puede pasar a EN EJECUCIÓN o no, esto dependerá de un thread propio de Java denominado Dispatcher, que es el encargado de otorgarle a un thread el recurso del procesador para que pueda ejecutarse.

Una vez EN EJECUCIÓN procesará su código hasta que se le acabe el tiempo asignado (también llamado Quantum) por la JVM o bien hasta que termine (si le da tiempo), o también se puede dar el caso de que alguien lo elimine mediante **stop()**. Los threads pasarán a ejecución dependiendo de sus prioridades y de acuerdo a la implementación de la JVM.

Una vez EN EJECUCIÓN los threads pueden, o pasar a LISTO de nuevo (si se acaba el quantum) o pasar a otros estados, a saber: EN ESPERA, DORMIDO, SUSPENDIDO y BLOQUEADO. Esto dependerá de la ejecución de ciertos métodos sobre el Thread (o la ocurrencia de ciertos sucesos).

Un Thread pasará a DORMIDO cuando se invoque el método sleep(), permanecerá así (sin consumir recursos) hasta que se le acabe el tiempo de "siesta", momento en el que volverá a LISTO. En este estado, el thread no consume recursos, es decir, no es candidato a serle asignado el procesador hasta que no despierte.

El Thread pasará a BLOQUEADO cuando tenga que sufrir una espera debida a E/S, saldrá de este estado en cuanto termine la E/S (tampoco consume recursos mientras espera).

El estado SUSPENDIDO es para aquellos Threads que han sufrido la invocación del método suspend(), en este estado permanecerán hasta que alguien los llame mediante resume(). Por supuesto tampoco consumen recursos en este estado.

Y por último el Thread pasará a EN ESPERA cuando alguien invoque un wait(), entonces el Thread pasará a esperar en el pool de threads. Esto implica que la próxima vez que se ejecute un notify(), el Thread se despertará.

También podemos ejecutar notifyAll(), de forma que sacamos a todos del pool.

Resumiendo, un thread entra en el estado "No Ejecutable" cuando ocurre uno de estos cuatro eventos:

- Alguien llama a su método sleep().

- Alguien llama a su método suspend().

- El thread utiliza su método wait() para esperar una condición variable.

- El thread está bloqueado durante la I/O.

Esta lista indica la ruta de escape para cada entrada en el estado "No Ejecutable":

- Si se ha puesto a dormir un thread, debe pasar el número de milisegundos especificados.

- Si se ha suspendido un thread, alguien debe llamar a su método resume().

- Si un thread está esperando una condición variable, siempre que el objeto propietario de la variable renuncie mediante notify() o notifyAll()

- Si un thread está bloqueado durante la I/O, cuando se complete la I/O.

Un thread puede morir de dos formas: por causas naturales o siendo asesinado (parado). Una muerte natural se produce cuando su método run() sale normalmente. Se puede matar un thread en cualquier momento llamando a su método stop().

El método stop() provoca una terminación súbita del método run() del thread. Si el método run() estuviera realizando cálculos sensibles, stop() podría dejar el programa en un estado inconsistente. Normalmente, no se debería llamar al método stop().

Scheduling

Java tiene un **Scheduler**, una lista de procesos, que monitoriza todos los hilos que se están ejecutando en todos los programas y decide cuales deben ejecutarse y cuales deben encontrarse preparados para su ejecución. Hay dos características de los hilos que el *scheduler* identifica en este proceso de decisión. Una, la más importante, es la prioridad del hilo de ejecución; la otra, es el indicador de *demonio*. La regla básica del *scheduler* es que si solamente hay *hilos demonio* ejecutándose, la *Máquina Virtual Java* (JVM) concluirá. Los nuevos hilos heredan la prioridad y el indicador de demonio de los hilos de ejecución que los han creado. El *scheduler* determina qué hilos deberán ejecutarse comprobando la prioridad de todos ellos, aquellos con prioridad más alta dispondrán del procesador antes de los que tienen prioridad más baja.
El *scheduler* puede seguir dos patrones, *preemptivo* y *no-preemptivo*. Los *schedulers preemtivos* proporcionan un segmento de tiempo a todos los hilos que están corriendo en el sistema. El

scheduler decide cual será el siguiente hilo a ejecutarse y llama al método *resume()* para darle vida durante un período fijo de tiempo. Cuando el hilo ha estado en ejecución ese período de tiempo, se llama a *suspend()* y el siguiente hilo de ejecución en la lista de procesos será relanzado (*resume()*). Los *schedulers no-preemtivos* deciden que hilo debe correr y lo ejecutan hasta que concluye. El hilo tiene control total sobre el sistema mientras esté en ejecución. El método *yield()* es la forma en que un hilo fuerza al *scheduler* a comenzar la ejecución de otro hilo que esté esperando. Dependiendo del sistema en que esté corriendo Java, el *scheduler* será de un tipo u otro, preemtivo o no-preemptivo.

Prioridades

El *scheduler* determina el hilo que debe ejecutarse en función de la prioridad asignada a cada uno de ellos. El rango de prioridades oscila entre 1 y 10. La prioridad por defecto de un hilo de ejecución es `NORM_PRIORITY`, que tiene asignado un valor de 5. Hay otras dos variables estáticas disponibles, que son `MIN_PRORITY`, fijada a 1, y `MAX_PRIORITY`, que tiene un valor de 10. El método *getPriority()* puede utilizarse para conocer el valor actual de la prioridad de un hilo.

Hilos Demonio

Los hilos de ejecución *demonio* también se llaman *servicios*, porque se ejecutan, normalmente, con prioridad baja y proporcionan un servicio básico a un programa o programas cuando la actividad de la máquina es reducida.

Los hilos demonio son útiles cuando un hilo debe ejecutarse en segundo plano durante largos períodos de tiempo.Un ejemplo de *hilo demonio* que está ejecutándose continuamente es el recolector de basura (*garbage collector*). Este hilo, proporcionado por la Máquina Virtual Java, comprueba las variables de los programas a las que no se accede nunca y libera estos recursos, devolviéndolos al sistema.

Un hilo puede fijar su indicador de demonio pasando un valor `true` al método *setDaemon()*. Si se pasa `false` a este método, el hilo de ejecución será devuelto por el sistema como un hilo de usuario. No obstante, esto último debe realizarse antes de que se arranque el hilo de ejecución (*start()*). Si se quiere saber si un hilo es un hilo demonio, se utilizará el método *isDaemon()*.

Diferencia entre hilos y *fork()*

fork() en Unix crea un proceso hijo que tiene su propia copia de datos y código del padre. Esto funciona correctamente si no hay problemas de cantidad de memoria de la máquina y se dispone de una CPU poderosa, y siempre que se mantenga el número de procesos hijos dentro de un límite manejable, porque se hace un uso intensivo de los recursos del sistema. Los applets Java no pueden *lanzar* ningún proceso en el cliente, porque eso sería una fuente de inseguridad y no está permitido. Las aplicaciones y los applets deben utilizar hilos de ejecución.

La *multitarea pre-emptiva* tiene sus problemas. Un hilo puede interrumpir a otro en cualquier momento, de ahí lo de *pre-emptive*. Fácilmente puede el lector imaginarse lo que pasaría si un hilo de ejecución está escribiendo en un array, mientras otro hilo lo interrumpe y comienza a escribir en el mismo array. Los lenguajes como C y C++ necesitan de las funciones *lock()* y *unlock()* para antes y después de leer o escribir datos. Java también funciona de este modo, pero oculta el bloqueo de datos bajo la sentencia `synchronized`:

```
synchronized int MiMetodo();
```

Otro área en que los hilos son muy útiles es en los interfaces de usuario. Permiten incrementar la respuesta del ordenador ante el usuario cuando se encuentra realizando complicados cálculos y no puede atender a la entrada de usuario. Estos cálculos se pueden realizar en segundo plano, o realizar varios en primer plano (música y animaciones) sin que se dé apariencia de pérdida de rendimiento.

Planificación de Threads

Qué significa planificación

Un tema fundamental dentro de la programación multihilo es la planificación de los hilos. Este concepto se refiere a la política a seguir de qué hilo toma el control del procesador y de cuando. Obviamente en el caso de que un hilo está bloqueado esperando una operación de IO este hilo debería dejar el control del procesador y que este control lo tomara otro hilo que si pudiera hacer uso del tiempo de CPU.¿Pero qué pasa si hay más de un hilo esperando?.¿A cuál de ellos le otorgamos el control del procesador?.

Prioridades

Para determinar qué hilo debe ejecutarse primero, cada hilo posee su propia prioridad: un hilo de prioridad más alta que se encuentre en el estado listo entrará antes en el estado de ejecución que otro de menor prioridad.

Para establecer la prioridad de un thread se utiliza el método setPriority() de la siguiente manera:

```
h1.setPriority(10);     //Le concede la mayor prioridad
h2.setPriority(1);      //Le concede la menor prioridad
```

También existen constantes definidas para la asignación de prioridad estas son :

```
MIN_PRIORITY=1;
NORM_PRIORITY=5;
MAX_PRIORITY=10;
```

Las cuales se pueden utilizar de la siguiente manera:

```
h1.setPriority(Thread.MAX_PRIORITY); //Le concede la mayor prioridad
h2.setPriority(Thread.MIN_PRIORITY); //Le concede la menor prioridad
```

Los métodos notify() y notifyAll()

Los métodos **notify()** y **notifyAll()** deben ser llamados desde el thread que tiene bloqueado el objeto para activar el resto de threads que están esperando la liberación de un objeto. Un thread se convierte en propietario del bloqueo de un objeto ejecutando un método sincronizado del objeto.

Cuando un thread llama a **wait()** en un método de un objeto dado, queda detenido hasta que otro thread llame a **notify()** en algún método del mismo objeto.

Por ejemplo, vamos a suponer cuatro empleados que se encuentran con su jefe y lo saludan, pero sólo luego de que éste los salude primero.

```
public class Ejemplo {
```

```
	public static void main(String argv[]) {
		Saludo hola = new Saludo();
		Personal pablo = new Personal(hola, "Pablo", false);
		Personal luis = new Personal(hola, "Luis", false);
		Personal andrea = new Personal(hola, "Andrea", false);
		Personal pedro = new Personal(hola, "Pedro", false);
		Personal jefe = new Personal(hola, "JEFE", true);
		pablo.start();
		luis.start();
		andrea.start();
		pedro.start();
		jefe.start();
		try {
			pablo.join();
			luis.join();
			andrea.join();
			pedro.join();
			jefe.join();
		} catch (Exception e) { System.out.println(e); }
	}
}
```

Creamos ahora la clase Saludo que será el objeto (o recurso) compartido por todos:

```
class Saludo {
	public synchronized void esperarJefe(String empleado) {
		try {
			wait();
			System.out.println(empleado+"> Buenos dias jefe!");
		}catch (InterruptedException e) {
			System.out.println(e.toString());
		}
	}

	public synchronized void saludoJefe() {
		System.out.println("JEFE> Buenos dias!");
		notifyAll();
	}
}
```

Por último creamos la clase Personal:

```
class Personal extends Thread {
	String nombre;
	Saludo saludo;
	boolean esJefe;
	Personal (Saludo s, String n, boolean j) {
		nombre = n;
		saludo = s;
		esJefe = j;
	}
	public void run() {
		System.out.println("("+nombre+" llega)");
			if (esJefe)
			saludo.saludoJefe();
		else
			saludo.esperarJefe(nombre);
	}
}
```

Se utilizó **notifyAll()** en lugar de **notify()**, porque en el segundo caso sólo se notificaría al primer thread (el primer empleado en llegar) y no a los demás, que se quedarían en el **wait()**.

Esta es la diferencia principal entre **notifyAll()** y **notify()**, el **notifyAll()** saca a todos los objetos thread del estado en espera, mientras que el método **notify()** saca a un solo objeto thread (imposible determinar cual al menos que tengan prioridades seteadas) del estado en espera.

Como se ve en la salida, a pesar de que los empleados están en condiciones de saludar, no lo hacen hasta que no llega el jefe:

```
(Pablo llega)
(Luis llega)
(Andrea llega)
(Pedro llega)
(JEFE llega)
JEFE> Buenos días!
Luis> Buenos días jefe!
Pedro> Buenos días jefe!
Andrea> Buenos días jefe!
Pablo> Buenos días jefe!
```

La importancia de la sincronización

La programación concurrente puede dar lugar a muchos errores debido a la utilización de recursos compartidos que pueden ser alterados. Las secciones de código potencialmente peligrosas de provocar estos errores se conocen como secciones críticas.

En general los programas concurrentes deben ser correctos totalmente. Este concepto es la suma de dos conceptos distintos la corrección parcial (o safety) esto es que no entrará nunca en ningún mal estado y la corrección temporal (o liveness) esto es que terminará en un algún momento de tiempo finito. En estos programas por tanto hay que evitar que varios hilos entren en una sección crítica (exclusión mutua o mutex) y que los programas se bloqueen (deadlock).

Además los programas que no terminan nunca (programas reactivos) deben cumplir la ausencia de inanición (starvation) esto es que tarde o temprano todos los hilos alcancen el control del procesador y ninguno quede indefinidamente en la lista de hilos listos y también deben cumplir la propiedad de equitatividad (fairwess) esto es repartir el tiempo de la forma más justa entre todos los hilos.

Un monitor impide que varios hilos accedan al mismo recurso compartido a la vez. Cada monitor incluye un protocolo de entrada y un protocolo de salida. En Java los monitores se consiguen mediante el uso de la palabra reservada synchronized bien como instrucción de bloque o bien como modificador de acceso de métodos.

La keyword synchronized

Cuando se utiliza la palabra clave synchronized se está indicando una zona restringida para el uso de "Threads", esta zona restringida para efectos prácticos puede ser considerada un candado ("lock") sobre la instancia del objeto en cuestión.

Lo anterior implica que si es invocado un método synchronized únicamente el "Thread" que lo invoca tiene acceso a la instancia del objeto, y cualquier otro "Thread" que intente acceder esta misma instancia tendrá que esperar hasta que sea terminada la ejecución del método synchronized.

El modificador synchronized puede aplicarse sobre un método:

- de instancia, indicando entonces que sólo puede haber un thread ejecutando un método sincronizado sobre el objeto correspondiente en cada momento

- de clase, sólo un thread creado a partir de un objeto de la clase puede estar ejecutando un método sincronizado en cada momento

A continuación se presenta la utilización de la palabra clave synchronized:

```
public class MiClaseThread implements Runnable {
	...
	syncronized void m1( ) // Método de instancia sincronizado
	{...}
	syncronized static void m2( ) // Método de clase sincronizado
	{...}
	public void run()
	{
		m1();
		m2();
	}
}

….

	public static void main(String[] args) {
		MiClaseThread z1=new MiClaseThread();
		MiClaseThread z2=new MiClaseThread();
		Thread t1=new Thread(z1);
		Thread t2=new Thread(z1);
		Thread t3=new Thread(z2);
		t1.start();
		t2.start();
		t3.start();
	}
```

Los Hilos t1 y t2 tiene el objeto z1 sincronizado en todos sus métodos.

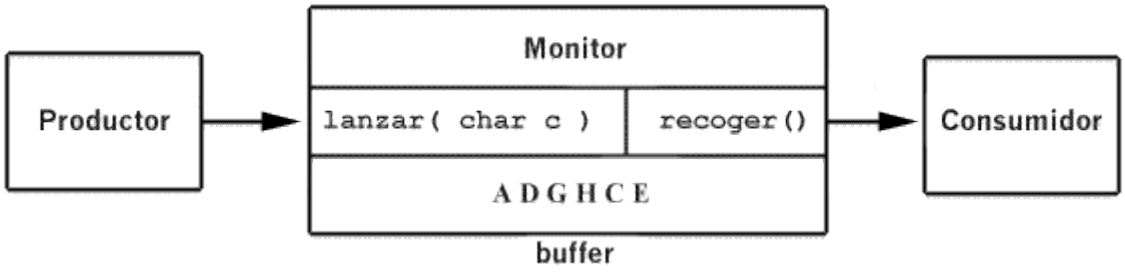
Y todos los hilos tiene el método m2 sincronizado.

Comunicación entre Hilos

Otra clave para el éxito y la ventaja de la utilización de múltiples hilos de ejecución en una aplicación, o aplicación *multithreaded*, es que pueden comunicarse entre sí. Se pueden diseñar hilos para utilizar objetos comunes, que cada hilo puede manipular independientemente de los otros hilos de ejecución.

El ejemplo clásico de comunicación de hilos de ejecución es un modelo productor/consumidor. Un hilo produce una salida, que otro hilo usa (consume), sea lo que sea esa salida. Entonces se crea un

productor, que será un hilo que irá sacando caracteres por su salida; y se crea también un *consumidor* que irá recogiendo los caracteres que vaya sacando el productor y un *monitor* que controlará el proceso de sincronización entre los hilos de ejecución. Funcionará como una tubería, insertando el productor caracteres en un extremo y leyéndolos el consumidor en el otro, con el monitor siendo la propia tubería.



Productor

El productor extenderá la clase **Thread**, y su código es el siguiente:

```
class Productor extends Thread {
    private Tuberia tuberia;
    private String alfabeto = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";

    public Productor( Tuberia t ) {
        // Mantiene una copia propia del objeto compartido
        tuberia = t;
    }

    public void run() {
        char c;

        // Mete 10 letras en la tubería
        for( int i=0; i < 10; i++ )
            {
            c = alfabeto.charAt( (int)(Math.random()*26 ) );
            tuberia.lanzar( c );
            // Imprime un registro con lo añadido
            System.out.println( "Lanzado "+c+" a la tuberia." );
            // Espera un poco antes de añadir más letras
            try {
                sleep( (int)(Math.random() * 100 ) );
            } catch( InterruptedException e ) {;}
         }
     }
}
```

Notar que se crea una instancia de la clase **Tuberia**, y que se utiliza el método ***tuberia.lanzar()*** para que se vaya construyendo la tubería, en principio de 10 caracteres.

Consumidor

Ahora se reproduce el código del consumidor, que también extenderá la clase **Thread**:

```
class Consumidor extends Thread {
    private Tuberia tuberia;
```

```
    public Consumidor( Tuberia t ) {
        // Mantiene una copia propia del objeto compartido
        tuberia = t;
    }

    public void run() {
        char c;

        // Consume 10 letras de la tubería
        for( int i=0; i < 10; i++ )
            {
            c = tuberia.recoger();
            // Imprime las letras retiradas
            System.out.println( "Recogido el caracter "+c );
            // Espera un poco antes de coger más letras
            try {
                sleep( (int)(Math.random() * 2000 ) );
            } catch( InterruptedException e ) {;}
        }
    }
}
```

En este caso, como en el del productor, se cuenta con un método en la clase **Tuberia**, *tuberia.recoger()*, para manejar la información.

Monitor

Una vez vistos el productor de la información y el consumidor, solamente queda por ver qué es lo que hace la clase **Tuberia**.

Lo que realiza la clase **Tuberia**, es una función de supervisión de las transacciones entre los dos hilos de ejecución, el productor y el consumidor. Los monitores, en general, son piezas muy importantes de las aplicaciones multihilo, porque mantienen el flujo de comunicación entre los hilos.

```
class Tuberia {
    private char buffer[] = new char[6];
    private int siguiente = 0;
    // Flags para saber el estado del buffer
    private boolean estaLlena = false;
    private boolean estaVacia = true;

    // Método para retirar letras del buffer
    public synchronized char recoger() {
        // No se puede consumir si el buffer está vacío
        while( estaVacia == true )
            {
            try {
                wait(); // Se sale cuando estaVacia cambia a false
            } catch( InterruptedException e ) { ; }
        }
        // Decrementa la cuenta, ya que va a consumir una letra
        siguiente--;
        // Comprueba si se retiró la última letra
        if( siguiente == 0 )
            estaVacia = true;
        // El buffer no puede estar lleno, porque acabamos
        // de consumir
        estaLlena = false;
```

```
        notify();

        // Devuelve la letra al thread consumidor
        return( buffer[siguiente] );
    }

    // Método para añadir letras al buffer
    public synchronized void lanzar( char c ) {
        // Espera hasta que haya sitio para otra letra
        while( estaLlena == true ) {
            try {
                wait(); // Se sale cuando estaLlena cambia a false
            } catch( InterruptedException e ) { ; }
        }
        // Añade una letra en el primer lugar disponible
        buffer[siguiente] = c;
        // Cambia al siguiente lugar disponible
        siguiente++;
        // Comprueba si el buffer está lleno
        if( siguiente == 6 )
            estaLlena = true;
        estaVacia = false;
        notify();
    }
}
```

En la clase **Tuberia** se pueden observar dos características importantes: los miembros dato (`buffer[]`) son privados, y los métodos de acceso (***lanzar()*** y ***recoger()***) son sincronizados.

Aquí se observa que la variable `estaVacia` es un semáforo, como los de toda la vida. La naturaleza privada de los datos evita que el productor y el consumidor accedan directamente a éstos. Si se permitiese el acceso directo de ambos hilos de ejecución a los datos, se podrían producir problemas; por ejemplo, si el consumidor intenta retirar datos de un buffer vacío, obtendrá excepciones innecesarias, o se bloqueará el proceso.

Los métodos sincronizados de acceso impiden que los productores y consumidores corrompan un objeto compartido. Mientras el productor está añadiendo una letra a la tubería, el consumidor no la puede retirar y viceversa. Esta sincronización es vital para mantener la integridad de cualquier objeto compartido. No sería lo mismo sincronizar la clase en vez de los métodos, porque esto significaría que nadie puede acceder a las variables de la clase en paralelo, mientras que al sincronizar los métodos, sí pueden acceder a todas las variables que están fuera de los métodos que pertenecen a la clase.

Se pueden sincronizar incluso variables, para realizar alguna acción determinada sobre ellas, por ejemplo:

```
sincronized( p ) {
    // aquí se colocaría el código
    // los threads que estén intentando acceder a p se pararán
    // y generarán una InterruptedException
}
```

El método ***notify()*** al final de cada método de acceso avisa a cualquier proceso que esté esperando por el objeto, entonces el proceso que ha estado esperando intentará acceder de

nuevo al objeto. En el método ***wait()*** se hace que el hilo se quede a la espera de que le llegue un ***notify()***, ya sea enviado por el hilo de ejecución o por el sistema.

Ahora que ya se dispone de un productor, un consumidor y un objeto compartido, se necesita una aplicación que arranque los hilos y que consiga que todos hablen con el mismo objeto que están compartiendo. Esto es lo que hace el siguiente trozo de código:

```
class java1007 {
    public static void main( String args[] ) {
        Tuberia t = new Tuberia();
        Productor p = new Productor( t );
        Consumidor c = new Consumidor( t );

        p.start();
        c.start();
    }
}
```

Compilando y ejecutando esta aplicación, se podrá observar en modelo que se ha diseñado en pleno funcionamiento.

Monitorización del Productor

Los programas productor/consumidor a menudo emplean monitorización remota, que permite al consumidor observar el hilo del productor interaccionando con un usuario o con otra parte del sistema. Por ejemplo, en una red, un grupo de hilos de ejecución productores podrían trabajar cada uno en una workstation. Los productores imprimirían documentos, almacenando una entrada en un registro (*log*). Un consumidor (o múltiples consumidores) podría procesar el registro y realizar durante la noche un informe de la actividad de impresión del día anterior.

Otro ejemplo, a pequeña escala podría ser el uso de varias ventanas en una workstation. Una ventana se puede usar para la entrada de información (el productor), y otra ventana reaccionaría a esa información (el consumidor).

Java Executor Service y Threading

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) { }
            }
```

```
	}
}
```

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.**

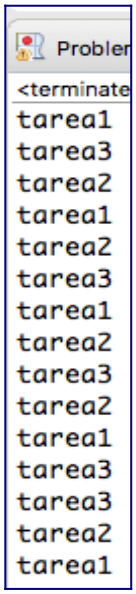
```
package com.arquitecturajava.executors;

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.

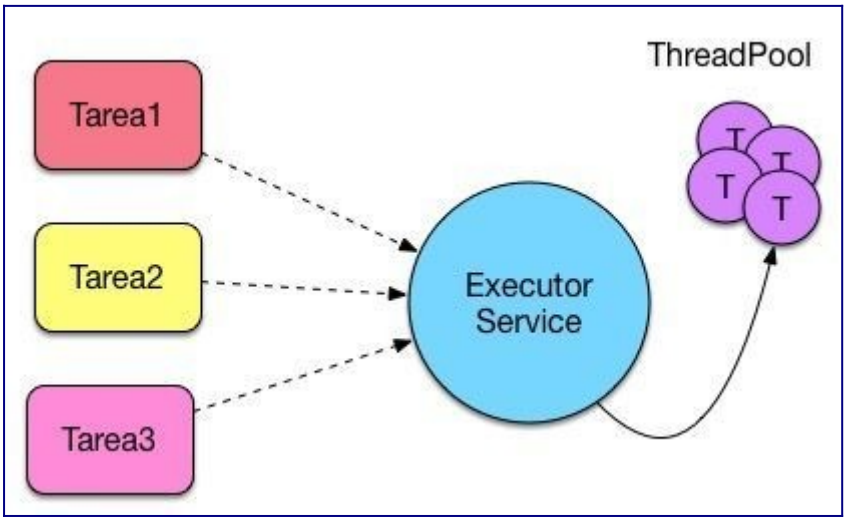
```
package com.arquitecturajava.executors;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
```

```
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 ejecutar el API cachearía un numero determinado de Threads y usaría solo estos.



La clase ExecutorService es muy útil cuando nos encontramos en situaciones de programación concurrente.

Java Callable Interface y su uso

Introducción

¿Para qué sirve un **Java Callable interface?.** Esta interface esta ligada 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:

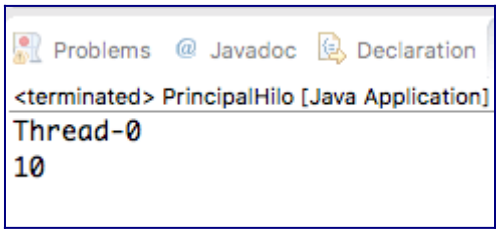
```
public class Tarea implements Runnable {
	@Override
	public void run() {
		int total = 0;
		for(int i=0;i<5;i++) {
```

```
                  total+=i;
                  try {
                        Thread.sleep(300);
                  } catch (InterruptedException e) { }
            }
            System.out.println(Thread.currentThread().getName());
            System.out.println(total);
      }
}
```

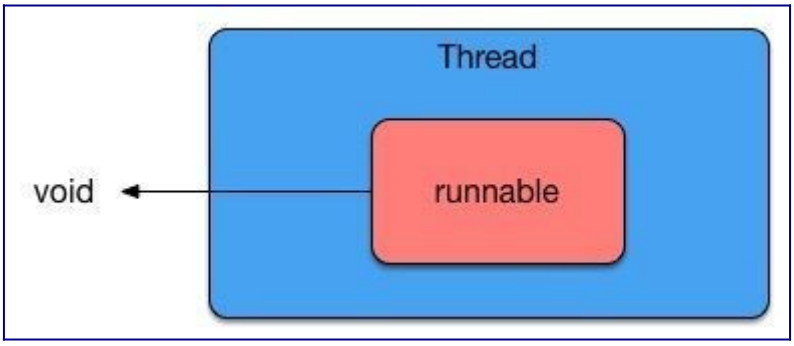
Acabamos de crear una tarea que implementa **la interface Runnable**. Podemos a partir de ella **crear un Thread** o hilo que la ejecute y nos imprima por pantalla la suma de los primeros 5 términos después de 1500 milisegundos (el bucle itera 5 veces) .

```
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 de la **interface Callable, vamos a verlo.**

```
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<=5;i++) {
			total+=i;
			try {
				Thread.sleep(300);
			} catch (InterruptedException e) { }
		}
		System.out.println(Thread.currentThread().getName());
		return total;
	}
}
```

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

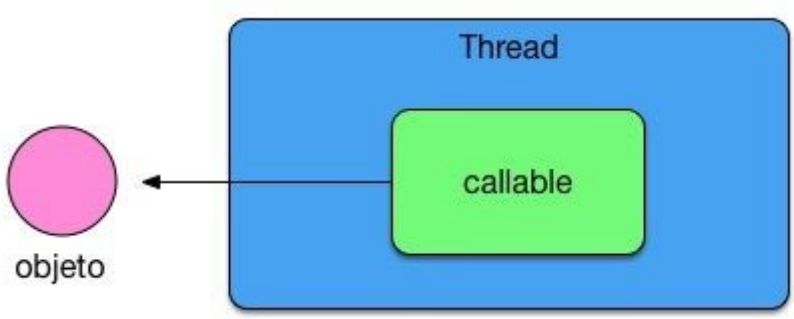
```
public void run();
public T call();
```

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

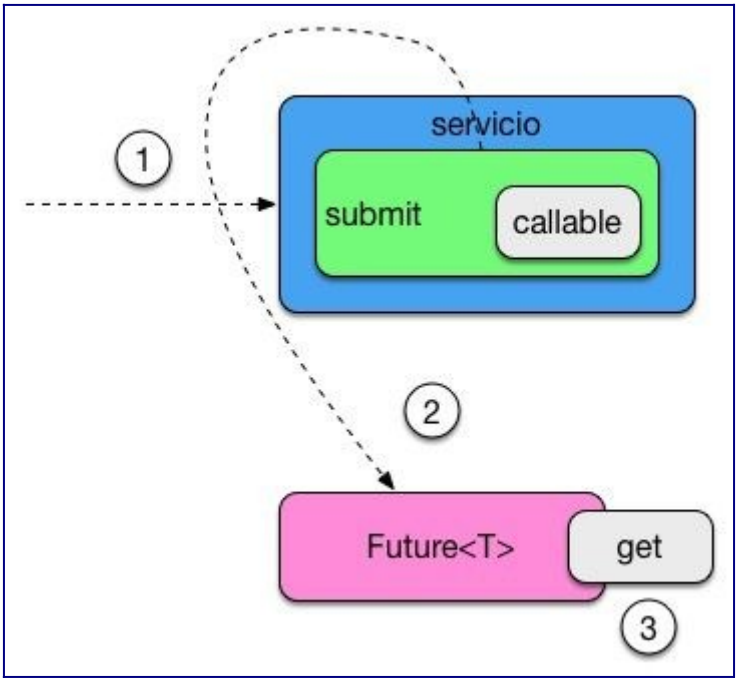
```
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;

public class PrincipalCallable {
	public static void main(String[] args) {
		try {
			ExecutorService servicio= Executors.newFixedThreadPool(1);
			Future<Integer> resultado= servicio.submit(new MiCallable());
			//resultado.isDone();
			//Devuelve un booleano pero no detiene el hilo de ejecución
			System.out.println("************************************");
			System.out.println(resultado.get());
			//Se detiene el hilo de ejecución hasta que este el resultado
			System.out.println("************************************");
			servicio.shutdown();
		} catch (InterruptedException e) {
			e.printStackTrace();
		} catch (ExecutionException e) {
			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 u**sando 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