Tema 2. Ampliación del libro

1 Introducción

En esta ampliación vamos a tratar tres asuntos importantes que van a ayudar a ampliar los conceptos de concurrencia Java que se ofrecen en el libro.

Lo primero será tratar los tipos de entornos multiprocesador concurrentes a los que nos podemos enfrentar. Seguiremos introduciendo los pool de hilos en Java como antecesores del ForkJoinPool paralelo del tema 3, la solución mas limpia que ofrece Java en entornos multiprocesador para hilos. La API Stream de Java se basa en esta solución para hacer paralelism0

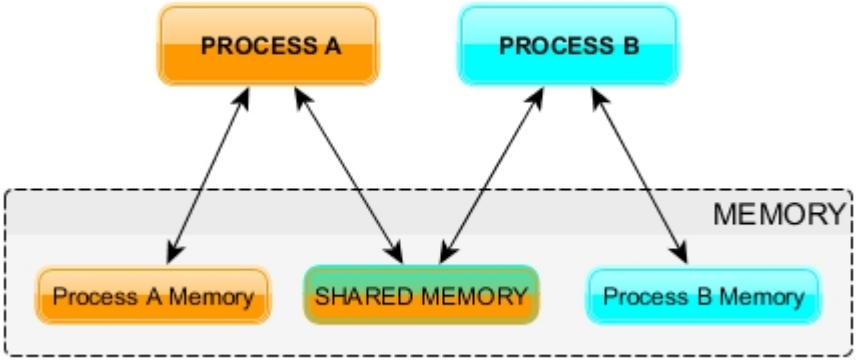
La tercera parte que introduciremos es el uso de las tareas callables y futures que se dejan listas para ejecutar a discreción del procesador. También nos servirá como antecesor para introducir la asincronía en Java con las Completable Futures.

2 Modelos de sincronización

Vamos a intentar explicar en detalle los modelos de sincronización utilizados por Java y vamos a recomendar que modelo usar dependiendo de que entorno.

Tenemos dos modelos de sincronización básicos en sistemas multihilo o multiprocesador:

El primero de ellos es el de memoria compartida. Lo hemos usado ya en el tema 2 creando las zonas de acceso sincronizado con synchronized. En este tema introduciremos los hilos atómicos

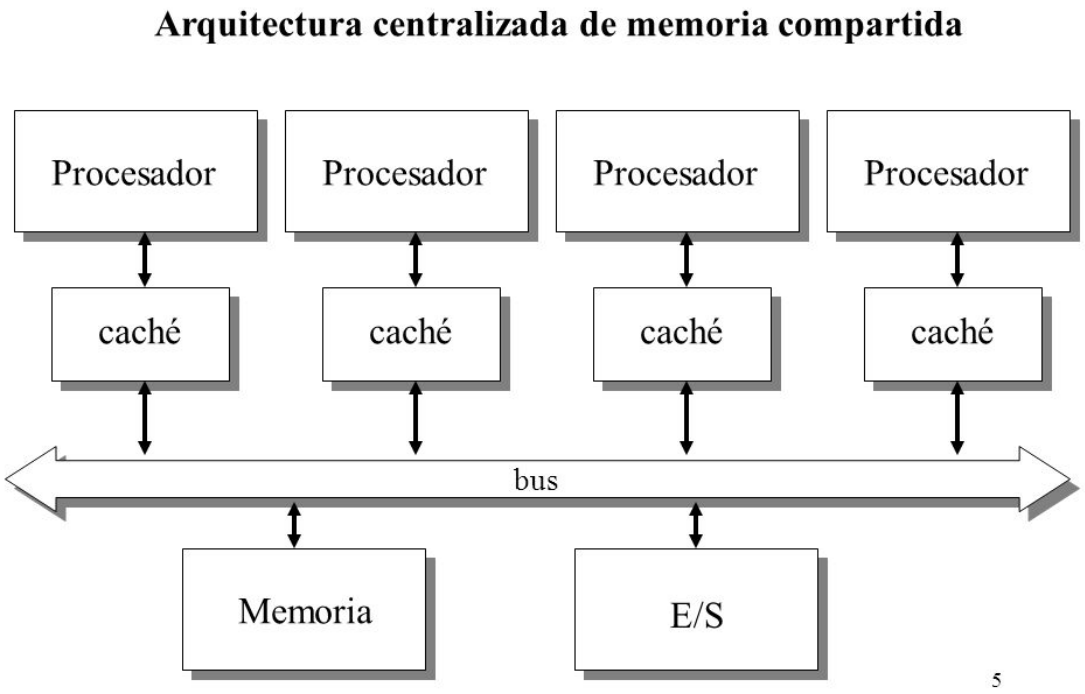


El segundo es el de paso de mensajes. En este modelo los hilos o procesos se sincronizan usando mensajes. En Java en el Tema 2 lo hemos utilizado con los métodos wait y notify.

2.1 Tipos de Sistemas multiprocesador. Soluciones de sincronización

Vamos a distinguir dos sistemas multiprocesador básicos que se usan habitualmente en entornos concurrentes.

Sistemas multiprocesador de memoria compartida. En estos sistemas todos los procesadores comparten una memoria común, aunque cada uno pueda tener su propia caché. Es el caso, por ejemplo de Java Hyperthreading de los Intel Core. Estos procesadores implementan instrucciones Lock para bloquear posiciones de memoria RAM. De esta manera cuando un proceso o hilo invoca a Lock/Unlock tenemos que el resto de los procesos o hilos que acceden a esa posición de memoria se queden bloqueados esperando a que la memoria sea liberada. Java implementa esta opción con las operaciones synchronize y con las operaciones de tipos Atómico que veremos en este tema.

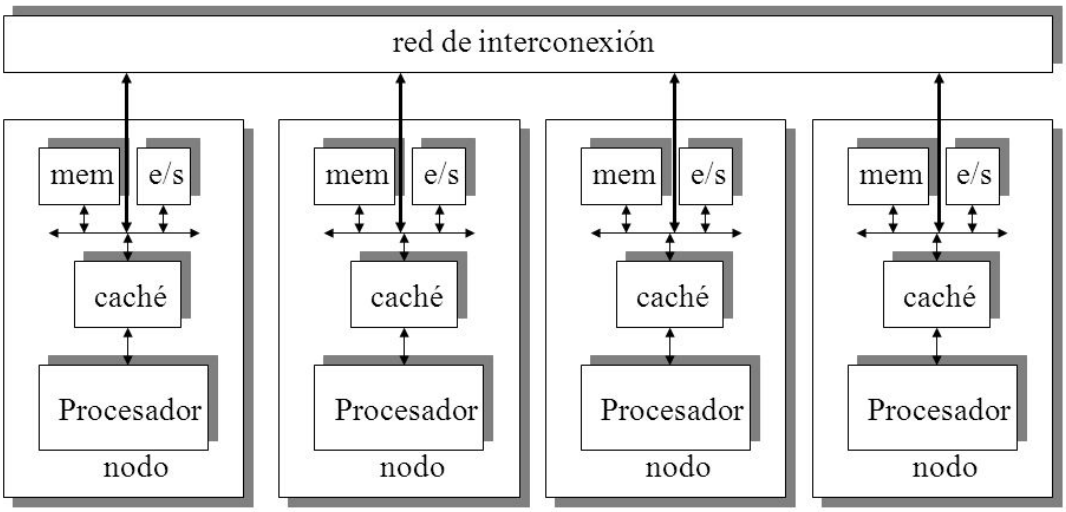


Reseñable también en Java que el modificador volatile obliga al compilador a que todas las operaciones sobre esa variable se hagan directamente en Memoria RAM además de en la cache. No se puede tener una versión más actualizada en el cache que en memoria RAM para esas variables. Igualmente todo lo que se hace sobre synchronize o tipos atómicos va directamente a memoria RAM como en el caso anterior.

Para resumir, en estos sistemas es muy recomendable el uso de sincronización con memoria compartida: Synchronize o tipos atómicos. Recordar la solución synchronize que vimos para el productor consumidor en este tema 2. Ofreceremos en la ampliación dos soluciones para el productor consumidor. Una con un tipo Atómico básico en Java, la otra con una cola Atómica que implementa Java llamada Blocking Queue.

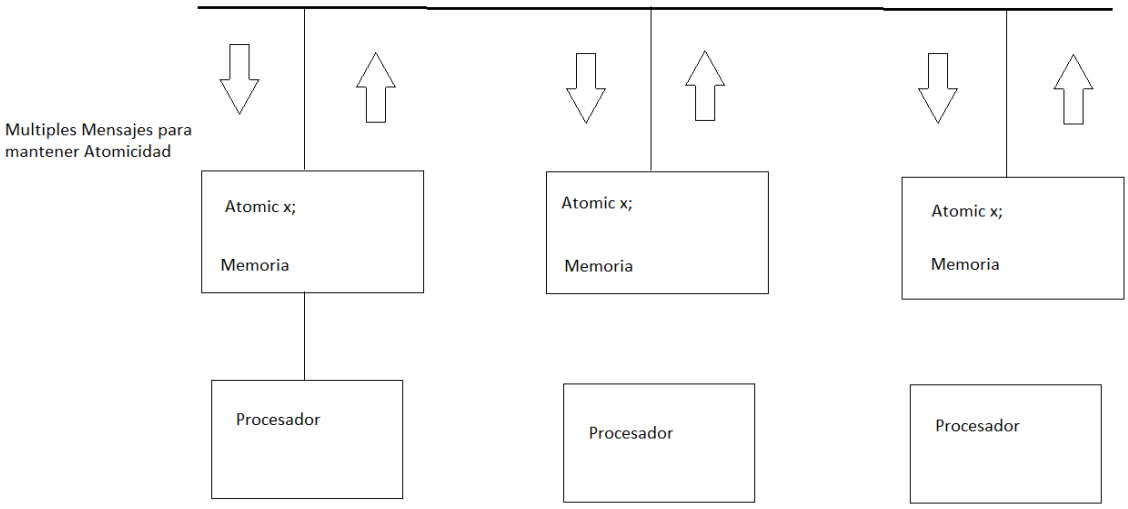
Sistemas procesadores de memoria distribuida, cada procesador tiene su memoria RAM propia. Los procesadores colaboran y se comunican entre ellos con paso de mensajes. Para comunicarse utilizan un bus o microred de altas prestaciones como podéis ver en la figura siguiente:

Arquitectura de memoria distribuida



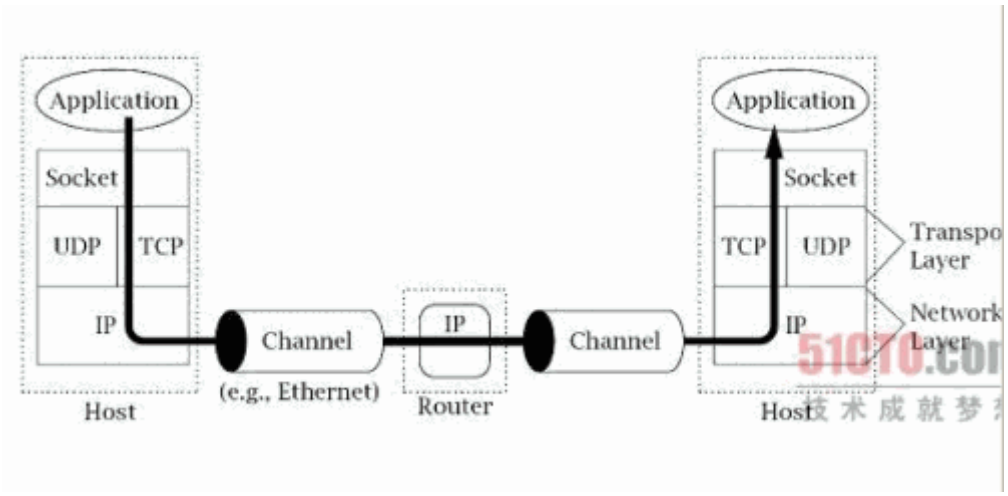
En estos sistemas si usamos en Java Synchronize o tipos atómicos la solución no es muy eficiente debido a que en realidad se hace en cada memoria individual de la variable compartida, de manera que todos los procesadores para mantener la atomicidad de la operación tienen que realizar múltiples mensajes. Imagina para un modelo de 256 procesadores la cantidad de mensajes que se deben generar usando synchronize.

En este tema vamos a ver el wait y el notify de Java para que los hilos se comuniquen entre ellos en memoria compartida bloqueándose para no consumir CPU. En estos entornos la sincronización entre hilos para resolver los problemas de exclusión mutua se debe hacer con wait y notify, como ya hemos visto en el tema 2. Se proporcionan un conjunto de ejemplos de comunicación entre hilos disponibles en el aula virtual



Soluciones posibles para este problema:

- **Para entornos distribuidos** usaremos **Java.net para sincronización**, en particular los Sockets y el protocolo TCP/IP que veremos en el siguiente tema son bastante útiles.
- Otra solución usando sockets muy eficiente que sea un procesador sólo el que mantenga el dato y reparta el acceso al resto. Este modelo que en programación se conoce como Objeto Activo lo explicaremos en el tema 3.



3 El Thread Pool

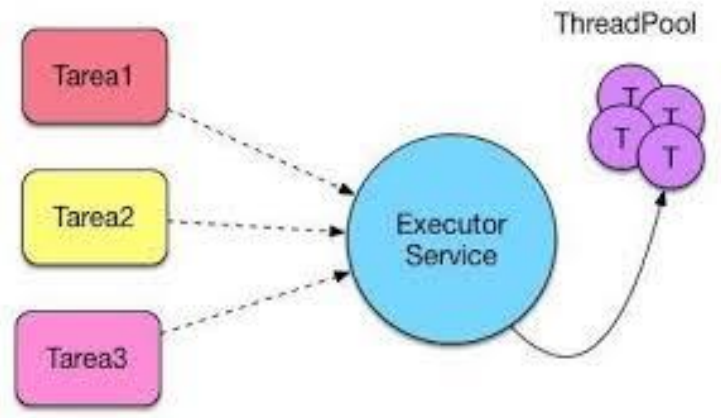
Vamos a realizar una **breve introducción al esquema de creación de hilos en la máquina virtual Java**, a partir de **java 5**. Son la base de la **ejecución concurrente y paralela** en la mayoría de las aplicaciones que tenemos instaladas en nuestros dispositivos, ya sean ordenadores, tables o móviles.

En Java, los hilos se asignan a hilos de nivel de sistema que son recursos del sistema operativo**. Si se crean hilos de forma incontrolable**, es posible que el ordenador se quede **sin estos recursos rápidamente**. Conduciría a un **funcionamiento lento y finalmente escalando** la situación a prácticamente **un bloqueo** de este. Todos los hilos mueren de inanición, ninguno usa los procesadores lo suficiente como para terminarse. El **cambio de contexto en sí**, se convierte en **una tarea extremadamente pesada** para cada procesador.

El **cambio de contexto entre hilos y tareas** también lo realiza el sistema operativo, con el fin de emular o implementar el paralelismo. Una manera simple de explicarlo es que **cuantos más hilos se generen**, **menos tiempo** pasa cada hilo **haciendo el trabajo real** en cada nucleo o procesador.

El patrón Grupo de hilos ayuda a **ahorrar recursos en una aplicación multiproceso** y también a **contener el paralelismo en determinados límites** predefinidos. La idea es **intentar limitar los errores** que se puedan cometer en programación teniendo un mayor control de los hilos en la máquina virtual java generado por nuestra aplicación.

Cuando se utiliza un grupo de hilo o tareas, **se escribe el código simultáneo en forma de tareas paralelas y se envían para su ejecución a una instancia de un grupo de hilos o tareas**. Esta instancia controla varios hilos para ejecutar dichas tareas.



Este patrón **permite controlar el número de hilos** que **la aplicación está creando,** su ciclo de vida, así **como programar la ejecución de los hilos** y mantener los hilos entrantes en una cola controladas.

4 Thread Pools en Java

4.1 Executors, Executor and ExecutorService

La **clase auxiliar *Executors*** contiene varios métodos para la **creación de instancias de un pool de hilos preconfigurado** automáticamente. Si **se usa, no se necesita aplicar ningún ajuste personalizado** en la creación de un pool de hilos. Es una ayuda que nos ofrece la **API de java**.

Los interfaces ***Executor*** y ***ExecutorService*** se utilizan para trabajar con diferentes **implementaciones del pool de hilos en Java**. La razón es que se debe mantener **el código del programador separado de la implementación real del grupo de hilos** y usar estas interfaces en toda la aplicación. Es una **barrera o mecanismo de protección que** ofrecen la mayoría de **frameworks para evitar efectos indeseados** en los programas. Permite mantener un **mayor control sobre nuestras ejecuciones**.

La interfaz Executor tiene un método ***de ejecución*** único **para enviar** instancias ***Runnable*** **para su ejecución. Es el método Execute.** En vez de lanzar los hilos directamente **con el método Start**, vamos **a usar esta diseño de la API de java que sigue el patrón ServiceProvider y Facade**, que nos proporcionará este mecanismo **de pool de hilos y de ejecución** a través de la librerias **java.util.concurrent**.

El siguiente **es un ejemplo rápido de cómo se puede usar la API *Executors*** para adquirir una **instancia de *Executor*** respaldada por un pool de un solo hilo. Se **ejecuta primero PrimerHilo**. Cuando termina se ejecuta **PrimerHiloR**.

En el ejemplo ejecutamos un **hilo que extiende de thread** y otro que **implementa Runnable**.

```
import java.util.concurrent.Executor;
import java.util.concurrent.Executors;

/**
 *
 * @author carlo
 */
public class IntroduccionAExecutors {


      public static void main(String[] args) {

            Executor executor = Executors.newSingleThreadExecutor();
```

```
            executor.execute(new PrimerHilo(5));
            executor.execute(new PrimerHiloR());




        }

}
```

PrimerHilo.java

```
public class PrimerHilo extends Thread {
        private int x;

        public PrimerHilo(int x) {
                 super("Hilo 1");

                this.x = x;
        }

        public void run() {
                for (int i = 0; i < x; i++)
                        System.out.println("En el Hilo con ácento... " +
i);
        }


}//PrimerHilo
```

PrimerHiloR.java

```
public class PrimerHiloR implements Runnable {
        public void run() {
                 System.out.println("Hola desde el Hilo! " +
                Thread.currentThread().getId());
        }




}
```

Executors: es una **clase de la API de java concurrent** que nos **ofrece métodos estáticos** para **obtener objetos de tipo Executor y ExecutorService**. Executor y ExecutorService son interfaces. **No sabemos exactamente que objetos reales nos está ofreciendo** la clase Executors.

Executor: es un **interfaz cuyos objetos ejecutan las tareas Runnable** que le son **enviadas, tanto Thread, hilos**, como **objetos que implemente Runnable**. Esta interfaz proporciona **una forma de envío de tareas** transparente al programador. Separamos la mecánica **de cómo se ejecutará cada tarea**, incluyendo detalles del uso de tareas, programación, etc. , del programa en si.

Normalmente **usamos un Executor para crear explícitamente hilos o tareas**. Por ejemplo, **en lugar de invocar new Thread**(new(RunnableTask())).start() para cada una de una de ellas, se usa **el método execute()** que te ofrece el interfaz.

https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/Executor.html

El interfaz de tipo ExecutorService implementa Executor. Nos ofrece la posibilidad de **ejecutar** además de objetos **de tipo Runnable**, objetos **de Tipo Callable**. Incluyendo la posibilidad de usar **Future y FutureTask,** para la ejecución de estas tareas como veremos más adelante. Igualmente nos ofrece el **método execute(), pero además aporta más métodos,** para la finalización de toda la estructura del pool de hilos.

4.2 Tipos de Pool

Executors nos ofrece diferentes **tipos de pool de hilos,** dependiendo del método que usemos. De esta manera **dispondremos de un sólo un hilo o tarea** para la ejecución de nuestras tareas, de **un número definido de hilos**, ó **de un pool que nos vaya generando nuevos hilos bajo demanda**, dependiendo del método que utilicemos. Aquí tenéis una tabla resumen del tipo de pools de hilos a usar.

Método para la creación de pools y factorias de hilos	**Tipos de pool y factorias**
newFixedThreadPool	**Pool con número fijo de hilos**. Cuando un hilo o tarea acaba de utilizar ese hilo**, el hilo vuelve a estar disponible** para ser usado por otro hilo o tarea
newSingleThreadExecutor	Sólo **tenemos un hilo o tarea** en el Pool. Sólo podemos ejecutar un hilo o tarea a la vez.
newCachedThreadPool	Se **cree un pool que va creando hilos a necesidad**. Si un hilo o tarea acaban tambien se reutilizan.
newSingleThreadScheduledExecutor **newFixedThreadScheduledPool** **newScheduledThreadPool**	Igual que **la versión normal pero permite ejecutar comandos** u otras **acciones periódicamente** o dado un espacio de tiempo.
defaultThreadFactory	Devuelve f**actoría para la creación** de **hilos**
privilegedThreadFactory	Devuelve **una factoría que crear hilos** con los **mismos permisos y privilegios** que **el proceso o hilo creador** de estos

4.3 Opciones de terminación con ExecutorService

La **interfaz *ExecutorService*** contiene un **gran número de métodos para controlar el progreso de las tareas** y administrar la **terminación del servicio de hilos**. Con esta interfaz, se puede enviar las tareas para su ejecución y también controlar su ejecución mediante la **instancia de objeto interface *Future* devuelta**. La clase que devuelve es **FutureTask**, que **implementa el interfaz Runnable** además de **Future**. También se ofrece **un servicio para la finalización tanto de las ejecuciones de tareas** e hilos, como de la eliminación del propio pool de hilos.

Cuando se termina de utilizar el **ExecutorService de Java**, debe terminarse, para que los hilos **no sigan ejecutándose**. Si la aplicación se inicia a través de un método **main()** y la ejecución principal sale de la aplicación, la **aplicación seguirá ejecutándose** si tiene un **ExecutorService activo** en la aplicación. Los **hilos activos dentro** de este **ExecutorService impiden** que la **JVM se apague**.

shutdown()

Para **terminar hilos y tares dentro de un ExecutorService**, se llama a su **método shutdown()**. **ExecutorService no se cerrará inmediatamente**, pero **ya no aceptará nuevas tareas** y, una vez que todos **los hilos hayan completado las tareas actuales, ExecutorService** se **cerrará**. Todas las tareas **enviadas a ExecutorService se ejecutan antes de llamar a shutdown().** A continuación, se muestra un ejemplo de cómo realizar un cierre de Java ExecutorService:

executorService.shutdown();

shutdownNow()

Si desea apagar **El Executor** inmediatamente, puede llamar al método **`shutdownNow()`**. Esto intentará detener todas las tareas o hilos **en ejecución de inmediato** y **omite todos los hilos o tareas enviadas**, pero **no procesadas**. No se proporcionan garantías sobre las tareas de ejecución. Tal vez se detengan, tal vez realicen la ejecución hasta el final.

```
executorService.shutdownNow();
```

awaitTermination()

El método awaitTermination(long timeout, TimeUnit unit) de **ExecutorService** bloqueará **el hilo de ejecución principal** que lo llama hasta que **ExecutorService** se cierre por completo o hasta que se **consuma un tiempo de espera determinado**. El primer parámetro **es un número indicando el tiempo** y el segundo **la unidad de tiempo** (TimeUnit.SECONDS , TimeUnit.MILISECONDS, TimeUnit.MINUTES, ETC) . El método **awaitTermination()** se llama típicamente después de llamar a **shutdown()** o **shutdownNow().** Este es un ejemplo de llamada a **ExecutorService awaitTermination()**

```
executorService.shutdown();

executorService.awaitTermination(5, TimeUnit.SECONDS);
```

executor2

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



            System.out.println("EJECUCION CON EXECUTOR Y VARIOS HILOS
EN EL POOL, SE EJECUTAN LOS DOS HILOS");
            Executor executor2 = Executors.newCachedThreadPool();
            executor2.execute(new PrimerHilo(5));
            executor2.execute(new PrimerHiloR());

      if ( !executor3.awaitTermination(5, TimeUnit.SECONDS)) {
                  executor3.shutdownNow();

             }

        }
```

Executor3

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



     ExecutorService executor3 = Executors.newFixedThreadPool(2);
                executor3.execute(new PrimerHilo(5));
           executor3.execute(new PrimerHiloR());


        if ( !executor3.awaitTermination(5, TimeUnit.SECONDS)) {
                  executor3.shutdownNow();

             }


        }
```

Executor4

```
En el siguiente ejemplo vamos a usar ExecutorService para ejecutar hilos.

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

```
            System.out.println("EJECUCION  CON  EXECUTORSERVICDE  Y  UN
VARIOS HILOS EN EL POOL EN EL POOL.NO ESPERAMOS");
            ExecutorService executor4= Executors.newCachedThreadPool();

            executor4.execute(new PrimerHilo(5));
            executor4.execute(new PrimerHiloR());



            if (!executor4.awaitTermination(5, TimeUnit.SECONDS)) {
                    executor4.shutdown();
                    System.exit(-1);
            }
            System.exit(1);



        }
```

¿Porque estamos usando System.exit(-1); y System.exit(1); en el ejemplo?

4.4 Ejercicio. (Cornell notes)

Executor	Notas
Executor2	En esta ejecución pasa … ¿Se ejecutan los hilos a la vez? Que pool usamos Porque … Con el shutdown Con el shutdownNow Además…
Executor3	
Executor4	

Executor5
Executor6
Executor7

4.5 Ejercicios

1. Crear un **nuevo programa HiloServiceBucle.java** que **tenga un bucle infinito**. Crear un clase **ejecutaHiloService** con un función **main** que cree un ExecutorService executor5. Probar el executor5.**awaitTermination**(5, TimeUnit.SECONDS)); con executor5.**shutdown**(); y **shutdownNow**().

2. Crear un **nuevo programa HiloServiceWait.java** que espere con un **wait y la clase SolicitaSuspender**. Crear un clase **ejecutaHiloServiceWait** con un función **main** que cree un ExecutorService executor5. Probar el executor6.**awaitTermination**(5, TimeUnit.SECONDS)); con executor6.**shutdown**(); y **shutdownNow**().

3. **Modifica el programa** del libro **MyHilo** correspondiente **a la actividad 2.4** del libro usando un ExecutorService llamado executor7. Probar executor7.**shutdown**(); y **shutdownNow**().

4. **Repasar el punto 2.3 y 2.4. Estudiad las ejecuciones** y sacar **vuestras conclusiones**. Explicaos con vuestras propias palabras **que hacemos en cada ejecución de los diferentes ExecutorService** y explicad porque **en función de los métodos** que usamos. Fijaos que tipo **de pool creamos**, si **esperamos o no esperamos**, etc.

4.6 Pool multiprocesador

Este pool se trata en detalle en el tema siguiente. Ahora vamos a introducirlo para utilizarlo con hilos solamente en este tema.En Java 8 y entornos Multiprocesador, la forma más **conveniente de obtener acceso a la instancia de *ForkJoinPool*** es utilizar su **método estático commonPool().** Como su nombre indica, esto **proporcionará una referencia al grupo común**, que es **un grupo de hilos predeterminado** para cada *ForkJoinTask*.

Según **la documentación de Oracle, el uso del grupo común predefinido reduce el consumo de recursos**, ya que esto **desalienta la creación de un grupo** de **hilos independiente por tarea.**

```
ForkJoinPool commonPool = ForkJoinPool.commonPool();
```

El mismo **comportamiento se puede lograr en Java 7 creando un ForkJoinPool** y asignándolo a **un campo estático público.**

```
public static ForkJoinPool forkJoinPool = new ForkJoinPool(2);
```

Pero **ahora en Java 8 y posteriores se puede acceder más fácilmente al pool** comun

```
ForkJoinPool forkJoinPool = PoolUtil.forkJoinPool;
```

Con los constructores de ForkJoinPool, es posible crear un grupo de **subprocesos personalizado con un nivel específico de paralelismo**, generador de hilos y controlador de excepciones. En el ejemplo anterior, el grupo tiene un nivel de paralelismo de 2. Esto significa que el grupo **utilizará 2 núcleos de procesador.**

```
new ForkJoinPool(2);
```

ForkJoinPool implementa java.util.concurrent.AbstractExecutorService, con lo que proporciona los mismos métodos que **hemos usado para ExecutorService**. Es el que **usaremos para los ejercicios de clase.**

En este primer ejemplo vamos a lanzar los hilos con **el método execute como en ExecutorService**. Con execute nos garantiza que el hilo se ejecutará en algún momento en el futuro.

```
pool.execute(h1);
```

Igualmente **esperamos a que terminen los hilos con el awaitTermination**. Esta implementación nos da más garantías de funcionamiento en terminación correcta que con el pool anterior executorService.

```
pool.awaitTermination(20, TimeUnit.SECONDS);
```

Finalmente si queréis terminar el proceso y sus hilos de manera abrupta, pero asegurando terminación del programa, podéis ejecutar, System.exit como vimos en el tema anterior, tanto si el awaitTermination arroja una excepción como si terminamos correctamente.

```
                try {
                pool.awaitTermination(20, TimeUnit.SECONDS);
            }
catch (InterruptedException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
                System.exit(-1);

            }

            System.exit(1);
```

```
package forkjoinpool;

import java.util.concurrent.ForkJoinPool;
import java.util.concurrent.TimeUnit;

public class HilosParalelos extends Thread {
	// constructor
	public HilosParalelos(String nombre) {
		super(nombre);
		System.out.println("CREANDO HILO PARALELO CADA UNO EN UN NUCLEO
DIFERENTE:" + getName());
	}

	// metodo run
	public void run() {
		for (int i=0; i<5; i++)
			System.out.println("Hilo:" + getName() + " C = " + i);
	}
```

```
	//
	public static void main(String[] args) {
		HilosParalelos h1 = new HilosParalelos("Hilo 1");
		HilosParalelos h2 = new HilosParalelos("Hilo 2");
		HilosParalelos h3 = new HilosParalelos("Hilo 3");

		ForkJoinPool pool= ForkJoinPool.commonPool();

		pool.execute(h1);
		pool.execute(h2);
		pool.execute(h3);


		System.out.println("3 HILOS INICIADOS...");


		try {
			sleep(10000);
		} catch (InterruptedException e) {
			// TODO Auto-generated catch block
			e.printStackTrace();
		}

			try {
			pool.awaitTermination(20, TimeUnit.SECONDS);
		} catch (InterruptedException e) {
			// TODO Auto-generated catch block
			e.printStackTrace();
			System.exit(-1);

		}

		System.exit(1);
	}// main

}// HiloEjemplo1
```

En el siguiente ejemplo vamos a lanzar los hilos como tareas. Es muy similar al anterior pero en teoría se espera resultado, porque se usa para tareas recursivas como veremos en el tema siguiente, donde se estudia este pool en profundidad.

Indicamos con esta creación de **pool que sólo usaremos 4 lineas de ejecución de nuestro sistema multiprocesador**.

```
ForkJoinPool pool = new ForkJoinPool(4);
```

Hacemos que los hilos se esperen unos a otros con el método join(). Hasta que no acaban todos, no acaba ninguno. El resto es practicamente igual que en el caso anterior con un detalle que preguntaremos.

```
h1.join();
h2.join();
```

```
h3.join();
```

```
package forkjoinpool;

import java.util.concurrent.ForkJoinPool;
import java.util.concurrent.TimeUnit;

public class HilosParalelos2 extends Thread {
	// constructor
	public HilosParalelos2(String nombre) {
		super(nombre);
		System.out.println("CREANDO HILO PARALELO CADA UNO EN UN NUCLEO
DIFERENTE:" + getName());
	}

	// metodo run
	public void run() {
		for (int i=0; i<5; i++)
			System.out.println("Hilo:" + getName() + " C = " + i);
	}

	//
	public static void main(String[] args) {
		HilosParalelos2 h1 = new HilosParalelos2("Hilo 1");
		HilosParalelos2 h2 = new HilosParalelos2("Hilo 2");
		HilosParalelos2 h3 = new HilosParalelos2("Hilo 3");

		ForkJoinPool pool = new ForkJoinPool(4);

		pool.submit(h1);
		pool.submit(h2);
		pool.submit(h3);


		System.out.println("3 HILOS INICIADOS...");


		try {
			h1.join();
			h2.join();
			h3.join();
		} catch (InterruptedException e) {
			// TODO Auto-generated catch block
			e.printStackTrace();
			System.exit(-1);

		}


		try {
			pool.awaitTermination(20, TimeUnit.SECONDS);
		} catch (InterruptedException e) {
			// TODO Auto-generated catch block
			e.printStackTrace();
			System.exit(-1);

		}
```

```
            if (pool.isTerminated())
                  System.exit(1);
            else
                  System.exit(-1);

      }// main

}
```

Como ultima pregunta del apartado por que se añade estas líneas con respecto al ejemplo anterior.

```
if (pool.isTerminated())
      System.exit(1);
else
      System.exit(-1);
```

5 Introducción a tipos atómicos

El **paquete java.util.concurrent.atomic** define **clases que admiten operaciones atómicas** en **variables de "tipos básicos"**. Todas **las clases tienen métodos get y set** que funcionan **como lecturas y escrituras** en **variables volátiles**. Es decir**, un conjunto tiene una relación sucede-antes con cualquier obtención posterior** en la misma **variable**. El **método compareAndSet atómico también tiene estas características de coherencia de memoria**, **al igual que los métodos aritméticos atómicos simples** que se aplican a variables atómicas enteras.

Para que os hagáis una idea, **este tipo de variables tiene un control temporal**, una marca de tiempo **para mantener la coherencia y el orden en que se hacen las operaciones**. Igualmente, las operaciones **sobre estas variables son atómicas**, sólo un hilo en **programación concurrente puede acceder a la vez a ellas**. Con este sistema **nos evitamos la sincronización** que es una tarea que ralentiza los programas java.

5.1.1 Solución implementada en Java para la exclusión mutua

Usando tipos atómicos resolvemos el problema de exclusión mutua que planteamos en el tema 1. En el siguiente ejemplo vamos a sustituir el tipo int

por AtomicInteger, como veréis en este caso la suma nos da 10000 por que las operaciones sobre la variable compartida de exclusión mutua.

```
package problemasconcurrencia;

import java.util.concurrent.atomic.AtomicInteger;

public class ExclusionMutuaAtomico {
	public static void main(String[] args) throws InterruptedException {
		class Counter {
			AtomicInteger count = new AtomicInteger( 0);

			public void increment() {
				count.incrementAndGet();
			}

			public int get() {
				return count.get();

			}
		}
		final Counter counter = new Counter();

		class CountingThread extends Thread {
			public void run() {
				for (int x = 0; x < 500000; x++) {
					counter.increment();
				}
			}
		}

		Integer numero=0;



		CountingThread t1 = new CountingThread();
		CountingThread t2 = new CountingThread();

		t1.start();
		t2.start();
		t1.join();
		t2.join();

		System.out.println("Con exclusión mutua debería dar 1000000");
		System.out.println("Al usar tipo atomico no violamos la exclusión
mutua el resultado es: " + counter.get());// deberia dar 1000000
	}
}
```

Tras ejecutar el ejemplo comprobareis que siempre devuelve el resultado correcto. ¿Por qué?

Nota: este modelo de programación con tipos atómicos es poco eficiente con sistemas multiprocesador de memoria distribuida, que quiere decir que cada procesador tiene su memoria propia y se sincronizan con paso de mensajes. Sin embargo, es muy eficiente con sistemas multiprocesador con memoria compartida, con vuestro Intel Core o AMD Razor multinúcleo.

Vamos a cambiar la solución para el productor consumidor haciendo contador atómico pero no con synchronize. Usaremos el

Para ver cómo se puede usar este paquete, **volvamos a la clase Contador que usamos** originalmente para **demostrar el efecto de los hilos en una clase sin control de concurrencia**. Ya hemos visto en clase, a **variable contador, de este ejemplo y la hemos sincronizado** para que **funcione de manera atómica.**

Solucion

```
class Contador {
    private int c = 0;

    public void incrementa() {
        c++;
    }

    public void decrementa() {
        c--;
    }

    public int getValor() {
        return c;
    }

}
```

La solución sincronizada que ya hemos realizado en clase sería esta:

```
class ContadorSincronizado {
    private int c = 0;
```

```
    public synchronized void incrementa() {
        c++;
    }

    public synchronized void decrementa() {
        c--;
    }

    public synchronized int getValor() {
        return c;
    }

}
```

Podemos mejorar esta solución usando tipos atómicos.

Para esta clase simple, la **sincronización es una solución aceptable**. Pero para una clase más complicada, es posible que **deseemos evitar el impacto de la sincronización innecesaria.** Reemplazar el **campo int** por un **AtomicInteger** nos permite evitar interferencias de subprocesos sin recurrir a la sincronización, como en **ContadorAtomico**::

```
import java.util.concurrent.atomic.AtomicInteger;

class AtomicCounter {
    private AtomicInteger c = new AtomicInteger(0);

    public void incrementa() {
        c.incrementAndGet();
    }

    public void decrementa() {
        c.decrementAndGet();
    }

    public int value() {
        return c.get();
    }

}
```

Solución con tipos atómicos. Probadlo. En la tabla del siguiente apartado tenéis los métodos explicados.

HiloAtomico.java

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.TimeUnit;
  import java.util.concurrent.atomic.AtomicInteger;
/**
 *
 * @author carlo
 */


class HiloAtomico extends Thread {

        ContadorAtomico c;
        public HiloAtomico(ContadorAtomico c, String nombre) {

            super(nombre);
            this.c=c;
        }

        @Override
        public void run() {
            for (int i=0 ; i<10 ; i++) {

                    c.incrementa();
                    System.out.println("  Hilo     "  +  this.getName()
+"Incrementa el contador" + c.valor());

                }



            }

    }
public class ContadorAtomico {



    private AtomicInteger c = new AtomicInteger(0);

    public void incrementa() {
        c.incrementAndGet();
    }

    public void decrementa() {
        c.decrementAndGet();
    }

    public int valor() {
        return c.get();
    }
```

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

      ContadorAtomico c = new ContadorAtomico();

      System.out.println("Contador atómico");



            ExecutorService              executorAtomico              =
Executors.newCachedThreadPool();


            executorAtomico.execute( new HiloAtomico(c,"Hilo 1"));
            executorAtomico.execute(new HiloAtomico(c,"Hilo 2"));




              if         (         !executorAtomico.awaitTermination(5,
TimeUnit.SECONDS)) {
                     executorAtomico.shutdownNow();

              }


  }

}
```

5.2 Tipos atómicos métodos soportados

Estos **son los métodos que podréis usar con los tipos atómicos**. Os doy los métodos relevantes para **AtomicInteger**, el resto son más o menos iguales. Al final de este apartado encontrareis el enlace a la referencia completa de tipos atómicos en Oracle.

Métodos para tipos atómicos. Ejemplos para AtomicInteger
`public final int get()` Obtiene el valor actual. **Devuelve:** el valor actual
`public final void set(int newValue)` Se asigna en el valor nuevo. **Parámetros:**

newValue - el nuevo valor
public final int getAndSet(int newValue) Atómicamente asigna el valor nuevo y devuelve el antiguo **Parámetros:** `newValue` – el nuevo valor **Devuelve:** el valor anterior
public final boolean compareAndSet(int expect, int update) Atómicamente asigna el valor actualizado , sie el valor esperado es igual al nuevo valor **Parameters:** `expect` – el valor esperado `update` – el nuevo valor **Devuelve:** verdadero si tiene éxito. La devolución falsa indica que el valor real no era igual al valor esperado.
public final int getAndIncrement() Incrementos atómicamente en uno el valor actual. **Devuelve:** el valor anterior
public final int getAndDecrement() Disminuye atómicamente en uno el valor actual. **Devuelve:** el valor anterior

Tenéis a **referencia completa a tipos atómicos en este enlace. Incluye entre otros AtomicBoolean, AtomicArrayInteger , AtomicReference, etc.**

AtomicArrayInteger nos permite hacer de arrays en java instancias atómicas. Muy útil para la programación concurrente. Pero en la actualidad se impone el **uso de la API Stream.**

AtomicReference, nos **permite lo mismo**, pero **con una referencia a un objeto como parámetro**. Se **usa en combinación con interfaces funcionales**, para realizar cambios sobre él. No podemos **ahondar en este tipo, porque no conocéis interfaces funcionales**.

Vamos a ver un ejemplo. Vamos a hacer que un objeto String se convierta en un tipo atómico. Se podría hacer con cualquier objeto. Os pongo el ejemplo con el Stream, para que os familiariceis con el asunto.

https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/atomic/package-summary.html

Ejemplo **ReferenciaAtomica.java**

```
import java.util.concurrent.atomic.AtomicReference;

public class ReferenciaAtomica {


        public     static     void     main(String[]     args)     throws
InterruptedException {


            String initialReference = "String referencia inicial";

            AtomicReference<String> atomicStringReference =
                new AtomicReference<String>(initialReference);

            String newReference = "String referencial nueva";
            boolean                    exchanged                    =
atomicStringReference.compareAndSet(initialReference, newReference);
            System.out.println("cambiamos referencia: " + exchanged);

            exchanged                                                =
atomicStringReference.compareAndSet(initialReference, newReference);
            System.out.println("cambiamos referencia: " + exchanged);

        }
}
```

Observar la siguiente ejecución: el método **`compareAndSet`** compara la **`initialReference`**, y el dato ya guardado en la **referencia atómica**. Si **son iguales devuelve** verdadero. Si son distintos **devuelve falso**.

Observar el resultado de la ejecución:

cambiamos referencia: **true**

cambiamos referencia: f**alse**

¿Porque nos devuelve dos valores diferentes? ¿Para que creéis que esta ese método?

5.3 Tipos atómicos en el problema del productor consumidor

Vamos a cambiar el problema del productor consumidor usnado tipos atómicos. La ventaja que nos **da el tipo atómico es que ya no necesitamos usar el wait y notify** con los hilos. Es muy sencillo, como **el tipo atómico ya está sincronizado** para que sólo un hilo pueda acceder a la vez no necesitamos **hacer el problemático wait y synchronized** que **puede bloquear el recurso** y hace **más lenta su ejecución**. En su lugar **sólo vamos a necesitar que la variable cambie de valor** para consumir o producir.

El **primer cambio es colocar la variable atómica booleana** en lugar de la variable tradicional. **Empezamos a false porque el productor** no ha producido nada todavía en la cola.

```
private AtomicBoolean disponible = new AtomicBoolean(false);
```

El método put es usado para que el productor produzca. Vamos a introducir cambios. Cuando **el productor hace un put**, asigna **true a la variable**, ahora el consumidor puede consumir.

```
        numero = valor;
        disponible.getAndSet(true);
```

Mientras **haya un valor disponible en la cola**, el **productor se queda esperando** a que **el consumidor**, consuma comprobando que disponible este a true en el bucle while (disponible.get()).

```
        System.out.println("Se produce: " + numero);
        while (disponible.get()) {
```

Ahora es **el turno del consumidor en el método get**, que es el que se usa para consumir. Inicialmente el consumidor se queda esperando a que el productor produzca en un bucle while (!disponible.get()), espera a que la variable disponible contenga el valor true. Cuando el productor ha producido, sale del bucle, consume, y pone la variable disponible a false. Le devuelve el turno al productor.

En el **get y el put dormimos durante un segundo** para que nos de tiempo a ver la **ejecución despacio**.

```
          disponible.getAndSet(false);
         return numero;
```

```
public class Cola {
    private int numero;
    private AtomicBoolean disponible = new
AtomicBoolean(false);//inicialmente cola vacia

    public  int get() {



         while (!disponible.get()) {

        try {
               sleep(1000);
           } catch (InterruptedException ex) {

Logger.getLogger(Cola.class.getName()).log(Level.SEVERE, null, ex);
           }
         System.out.println("esperando a productor " );
          }

          System.out.println("Se consume: " + numero);

           disponible.getAndSet(false);
         return numero;
       }
```

```
    public  void put(int valor) {

        numero = valor;
        disponible.getAndSet(true);
        System.out.println("Se produce: " + numero);
        while (disponible.get()) {

                System.out.println("esperando a consumidor " );

            try {
                sleep(1000);
            } catch (InterruptedException ex) {

Logger.getLogger(Cola.class.getName()).log(Level.SEVERE, null, ex);
            }

          }



        }



}
```

Esta **solución es mucho más eficiente y segura que la solución con el wait**, porque no nos hace falta **bloquear hilos con wait**. Además, **nos lo hace más fácil**, ya que sabemos que las **operaciones son atómicas**. Cuando se realicen, se realizarán en orden, y mientras **el hilo hace la operación no puede ser bloqueado**, hasta que **acabe esa operación**. Se **bloqueará después en el sleep**, pero **nunca con la operación sobre el tipo atómico a medio**. Es **imposible bloquear recursos de esta manera**.

6 Hilos con lambdas.

En versiones **anteriores a la versión 8 de Java, hay interfaces que ya pueden ser considerados como funcionales**. Se les conoce con el nombre **de interfaces Legacy**. Por ejemplo, los interfaces Runnable y Callable. Runnable es un interfaz que se usa para la programación de hilos y Callable para la programación de Tareas. Os dejo este ejemplo para que lo veáis. Pueden ser usados como **interfaces funcionales porque definen un solo método abstracto**. Se ajustan al estándar de interfaces funcionales.

En **este sencillo ejemplo os indico como muchos programadores usan los hilos** en la actualidad con **funciones anónimas, sobreescribiendo el método run en tiempo de ejecución**. Es muy parecido a lo que hacemos con los listeners. O con **expresiones lambda**.

En el primer hilo usamos una **expresión lambda** para crear un hilo new Thread(() ->

```
          executor2.execute(new              Thread(()              ->
System.out.println("Expresion  lambda,  interfaces  legacy,  expresion
lambda, sobreesribo el metodo run en tiempo de ejecucion")));
```

En el segundo hilo usamos una función anónima para sobreescribir el método run de un hilo.

En la tercera sobreescribimos el método run de un **objeto runnable Runnable.** Los **interfaces, interfaces funcionales, y legacy nos permiten crear objetos** de ellos directamente con un new.

```
            executor2.execute(new Runnable() {

                    @Override
                public void run () {


                    System.out.println("He    cambiado    PrimerHiloR
runnable en tiempo de ejecucion sobreescribiendo el método Run");

                }

            });
```

EjemploHiloJavaOcho.java

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.TimeUnit;

class HiloRunnable implements Runnable {
       public void run() {
               System.out.println("Hola desde el Hilo Runable! " +
              Thread.currentThread().getId());
       }
```

```
}

public class EjemploHiloJavaOcho {


    public static void main(String[] args) throws InterruptedException
{

         System.out.println("EJECUCION java 8 de hilos");
            ExecutorService                executor2                =
Executors.newCachedThreadPool();

           executor2.execute(new              Thread(()              ->
System.out.println("Expresion  lambda,  interfaces  legacy,  expresion
lambda, sobreesribo el metodo run en tiempo de ejecucion")));



            executor2.execute(new Thread("Hilo java 8") {

                @Override
                public void run () {

                    System.out.println(""
                            + "hilo  estandar  "  +  this.getName()  +
"Funcioón anónima. en tiempo de ejecucion sobreescribiendo el método
un");
                }



            });


            executor2.execute(new HiloRunnable() {

                    @Override
                public void run () {


                    System.out.println("He    cambiado    PrimerHiloR
runnable en tiempo de ejecucion sobreescribiendo el método Run");

                }

            });

         if ( !executor2.awaitTermination(5, TimeUnit.SECONDS)) {
                    executor2.shutdownNow();

              }
    }

}
```

6.1 Ejercicios

Ejercicio1. Introducir tipos atómicos y executor en el ejemplo CompartirInfo4, y en ProductorConsumidor.

Ejercicio2. Introducir tipos atómicos y executor en el ejemplo del ProductorConsumidor.

Ejercicio 3. Crear un hilo con una función anónima que sume los 100 primeros números pares.

7 Callable

La necesidad de Callable

Hay **dos formas de crear hilos**: una **extendiendo la clase Thread** y otra mediante la creación de **un hilos con un Runnable**. Sin embargo, una característica que falta en Runnable es que no **podemos hacer que un resultado devuelto** de un hilo o tarea cuando finaliza, es decir, cuando run() se completa. Para soportar esta característica, la interfaz Callable está presente en Java.

Callable vs Runnable

- Para **implementar Runnable, es necesario implementar el método run()** que no devuelve nada, mientras que para un **método Callable, el call()** debe implementarse, lo que devuelve **un resultado al finalizar**. Se ha de tener en cuenta que **un hilo no se puede crear con un Callable**, solo se puede crear con un Runnable.
- Otra diferencia es que **el método call() puede producir** una **excepción** mientras **que run()** no puede.

7.1 Future

Cuando se completa el método call(), la respuesta debe **almacenarse en un objeto conocido por el hilo principal**, para que el **hilo o función principal pueda conocer el resultado que el hilo llamado** devolvió. ¿Cómo **almacenará el programa y obtendrá** este **resultado** más adelante? Para ello, **se puede utilizar un objeto que implemente el interfaz Future.** Piensa en una instancia de

Future como un objeto que contiene el resultado - puede que no lo tenga en este momento, pero lo hará en el futuro (una vez que el objeto Callable lo devuelve). Por lo tanto, un futuro es básicamente **una forma en que el hilo o ejecución principal** puede realizar un **seguimiento del progreso y el resultado de otros hilos**.

Observe que Callable y Future hacen dos cosas diferentes: **Callable es similar a Runnable**, en el que encapsula una tarea que está destinada a ejecutarse en otro hilo, mientras que **un objeto Future se usa para almacenar un resultado** obtenido de **un hilo diferente**. De hecho, **los objetos Future** también se puede hacer para **trabajar con Runnable**.

- **public boolean cancel(boolean mayInterrupt):** Se utiliza para detener la tarea. Detiene la tarea si no se ha iniciado. Si se ha iniciado, interrumpe la tarea solo si mayInterrupt es true.
- **public Object get() produce InterruptedException, ExecutionException:** se utiliza para obtener el resultado de la tarea. Si la tarea se ha completado, devuelve el resultado inmediatamente, de lo contrario espera hasta que se complete la tarea y, a continuación, devuelve el resultado.
- **public boolean isDone():** Devuelve true si la tarea está completa y false de lo contrario

Para **crear el hilo, se requiere un Runnable**. Para **obtener el resultado, se requiere un Objeto Future**. La clase que implementa **ambos interfaces es FutureTask**.

Java tiene **el tipo concreto FutureTask**, que implementa **Runnable y Future, combinando ambas funciones convenientemente**. Un **FutureTask se puede crear proporcionando a su constructor con un Callable**. A continuación, el objeto **FutureTask** se **le proporciona al constructor de Thread** para **crear el objeto Thread**, y **lanzar la ejecución de esa tarea**. Por lo tanto, indirectamente, **el hilo se crea con un Callable**. **No hay manera de crear la tarea** directamente con un Callable**. Lo veremos más adelante**. Debemos usar el framework que nos ofrece la API de java Concurrent.

7.1.1 Ejemplo 1

Vamos a contruir una tarea, clase Java, **EsPrimo** que con un objeto **Callable PrimoCallable** y un Objeto **Future** nos diga si un número es primo o no. Definimos una clase interna **class PrimoCallable implements Callable**. Al implementar el interfaz Callable debemos **sobreescribir el método call()**. El objeto Callable debe devolvernos un booleano diciendo si es verdadero o falso que el número es primo. El número se pasa al constructor del objeto Callable.

```
public PrimoCallable(int numero) {
        this.numero = numero;
    }
```

La función call del objeto Callable es la que nos tiene que **devolver verdadero o falso**. Se ejecutará cuando llamemos al método get del objeto Future al que asociaremos este objeto Callable.

```
public Boolean call() throws Exception {


        return esPrimo(numero);
    }
```

La tarea esPrimo asocia el objeto Callable a un Objeto Future con la función submit.

Future<Boolean> f1 = executor.submit(new PrimoCallable(numero));

Fijaos que **en el objeto Future tenemos** que **indicar** que el tipo que devuelve el **PrimoCallable es Boolean**. Con el submit dejamos la tarea Callable preparada para ejecutarse, pendiente de ejecución. Pasara a **lista para ejecutarse** cuando llamemos al método **get** del **objeto Future f1**.

```
if (f1.get()) {
                System.out.println("El numero " + numero +  " es primo"
);

               } else {

                System.out.println("El  numero  "  +  numero  +   "  no  es
primo" );
}

                    executor.shutdown();
```

Si **el número pasado es primo el f1.get() devolverá true**. En caso contrario, devolverá false. El **programa espera a que la tarea termine**. Cuando termina debemos cerrar nuestro pool de hilos con el método ShutDown().

Nota: usar siempre clase, no tipos básicos, Integer en lugar de int. Procurar meter los ejemplos en paquetes para asegurarnos que funcionan

TareaEsPrimo.java

```
import java.util.List;
import java.util.Scanner;
import java.util.concurrent.Callable;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;


   class PrimoCallable implements Callable {
    private int numero;

    public PrimoCallable(int numero) {
        this.numero = numero;
    }



      public static boolean esPrimo(int n)
        {
            boolean continuar = true;
               boolean esPrimo = true;
               long divisor = 2;
               do {
                       if (n % divisor == 0) {
                               continuar = false;
                               esPrimo = false;
                       } else
                               ++divisor;

               } while (continuar && divisor <= (n/2));

               return esPrimo;
      }

    @Override
    public Boolean call() throws Exception {


        return esPrimo(numero);
    }

    }
/**
 *
 * @author carlo
 */
public class TareaEsPrimo {
```

```
    public TareaEsPrimo () {



    }

    public void EjecutarTarea() {

          Integer numero=0;
          Scanner in = new Scanner(System.in);

         System.out.println("Dame un número por pantalla");


         numero= in.nextInt();

         ExecutorService executor = Executors.newSingleThreadExecutor();
         Future<Boolean>         f1         =         executor.submit(new
PrimoCallable(numero));

          try {
                        if (f1.get()) {
                            System.out.println("El numero " + numero +  "
es primo" );

                          } else {

                           System.out.println("El numero " + numero +  "
no es primo" );
                          }

                } catch (InterruptedException | ExecutionException e) {

                        e.printStackTrace();
                }

                     executor.shutdown();
    }

    public static void main(String[] args) {


             TareaEsPrimo tarea =new TareaEsPrimo();
             tarea.EjecutarTarea();
    }

}
```

En el siguiente ejemplo lo veréis más claro. Usamos ExecutorService para lanzar la tarea Callable con un Future. Llamando a submit.

Executors.newCachedThreadPool(); Nos da un número ilimitado de hilos que se crean dinámicamente.

ExecutorService executor = Executors.newCachedThreadPool();

Usamos la clase **ExecutorService** para ejecutar una tarea **Callable** en combinación con **Future**.

```
Future<List<Integer>>          f1          =          executor.submit(new
FutureCallableEjemplo1(8));
```

En el submit de la clase ExecutorService lanzamos la tarea como pendiente de ejecución. El resultado se recogerá en el objeto Future. Como veis para el objeto Future**<List<Integer>>** hay que declarar el tipo de vuelta, una lista de enteros **List<Integer>.**

La tarea se realizará cuando ejecutemos el método **get()** del Objeto Future. `f1.get();` devuelve una lista con 8 números. Cuando llamo a get, la tarea pasa a estado lista para ejecutarse. Cuando la tarea empiece a ejecutarse se llamará al método de Callable public **List<Integer> call()** . Como podéis observar, devuelve una lista de enteros.

Cuando termine se imprimirá la lista por pantalla con el println.

```
                try {
                  System.out.println("Resultado tarea 1 lista numeros"
+ f1.get());
                  System.out.println("Resultado tarea 2 lista numeros"
+ f2.get());
            } catch (InterruptedException | ExecutionException e) {
                  e.printStackTrace();
            }
```

TareaFutureCallableEjemploListas.java

```
import java.util.Arrays;
import java.util.List;
import java.util.concurrent.ExecutionException;
```

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

/**
 *
 * @author carlo
 */
public class TareaFutureCallableEjemploListas {

     public TareaFutureCallableEjemploListas() {
         // Sólo se ejecuta una tarea en un hilo
                 ExecutorService              executor              =
Executors.newCachedThreadPool();


                 Future<List<Integer>>    f1    =    executor.submit(new
FutureCallableEjemploListas(8));
               Future<List<Integer>>    f2    =    executor.submit(new
FutureCallableEjemploListas(12));


          try {
               System.out.println("Resultado tarea 1 lista numeros" +
f1.get());
               System.out.println("Resultado tarea 2 lista numeros" +
f2.get());
               } catch (InterruptedException | ExecutionException e) {

                       e.printStackTrace();
               }

                    executor.shutdown();
           }



     public static void main(String[] args) {


            new TareaFutureCallableEjemploListas();

    }
```

FutureCallableEjemploListas.java

```
import static java.lang.Thread.sleep;
import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.Callable;

/**
 *
```

```
 * @author carlo
 */
public class FutureCallableEjemploListas implements Callable {
     List<Integer> numeros;
     int tamanyo;

    public FutureCallableEjemploListas(Integer tam) {



             tamanyo = tam;
             numeros = new ArrayList(tam);


    }

       @Override
        public List<Integer> call() throws Exception {

            for(int i=0;i<tamanyo; i++ ) {

                numeros.add(i);
            }

            sleep(1000);

            return  numeros;
    }

}
```

7.1.2 Ejercicios

1. Realizar un **programa que devuelva una lista de números** aleatorios de tamaño n. El **tamaño de la lista, n** será pasado como parámetro.

2. Realizar otro programa principal **NPrimo.java** que con un callable y un future **devuelva el n número primo**. Por ejemplo si le paso al programa un 6 me devolverá el número 17.

   [$^{1}3$, $^{2}5$, $^{3}7$, $^{4}11$, $^{5}13$, $^{6}17$, $^{7}19$, $^{8}23$, $^{9}29$, $^{10}31$, $^{11}37$, $^{12}41$, $^{13}43$,…, ^{n}Xn]

3. Realizar un programa principal **ListaNPrimos.java** que dado un número n me devuelva una lista con los n primeros números primos. Por ejemplo, si le paso un 12 me devolvería esta lista con los 12 primeros números primos.

   [3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43]

8 Future y runnables

Vamos a usar Future ahora para **lanzar objetos Runnables**. El concepto es el mismo que en el caso anterior. Pero en este caso runnable no devuelve nada. Para ello vamos a usar en este caso la clase **FutureTask** que **implementa** los interfaces **Runnable y Future**

9 Future Tasks

9.1 Future:

Como ya hemos explicado, una interfaz **Future** proporciona métodos para **comprobar si el cálculo se ha completado, esperar su finalización y recuperar los resultados del** cálculo. El resultado se recupera utilizando **el método get() de Future** cuando el cálculo se ha completado, y se bloquea hasta que se completa.

Future y FutureTask están disponibles en **el paquete java.util.concurrent** de Java 1.5.

9.2 FutureTask:

FutureTask es la clase que implementa el interfaz Future. Es el objeto que te devuelve el método de ExecutorService **submit().** Vamos a ver en este apartado como **además de esperar a que un Callable acabe, podemos ejecutar un hilo o Runnable con un Future o FutureTask y esperar a que acabe igualmente.**

Básicamente **se puede usar con Runnables porque los Runnables no devuelven nada** al **ejecutar el método run**. En una **declaración de FutureTask** como veréis

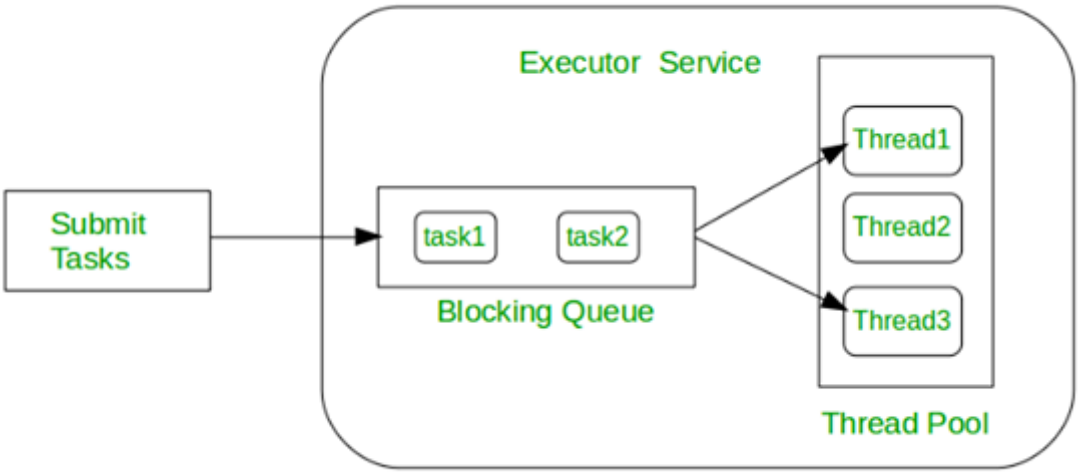
a continuación puedo omitir el tipo del objeto FutureTask, que no se devuelva nada.

```
Future<String>

        futureTask1 = new FutureTask<>(miRunnable1,

                                       "FutureTask1 completa");
```

Un ejemplo de uso de Future es trabajar con **grupos de hilos.** Cuando se envía **una tarea a ExecutorService** que **tarda mucho tiempo de ejecución**, devuelve un objeto Future inmediatamente. Este objeto Future **se puede utilizar para que cuando se completa la tarea se devuelve el resultado del cálculo.**



9.3 Ejecución de runnables como tareas

Ejemplos: en el siguiente ejemplo se crean dos Tareas. Cuando se termina de ejecutar la primera **a los dos segundos se ejecutará la segunda**. Podría hacerlo con **el interfaz Future como en el ejemplo anterior.**

`EjemploFutureTask.java`

```
Ejemplo
```

```
import java.util.concurrent.*;
import java.util.logging.Level;
import java.util.logging.Logger;

class MiRunnable implements Runnable {

    private final long tiempoDeEspera;

    public MiRunnable(int tiempoEnMilis)
    {
        this.tiempoDeEspera = tiempoEnMilis;
    }

    @Override
    public void run()
    {
        try {
            // sleep for user given millisecond
            // before checking again
            Thread.sleep(tiempoDeEspera);

            // return current thread name
            System.out.println(Thread
                                    .currentThread()
                                    .getName());
        }

        catch (InterruptedException ex) {
            Logger
                .getLogger(MiRunnable.class.getName())
                .log(Level.SEVERE, null, ex);
        }
    }
}

class EjemploFutureTask {

    public static void main(String[] args)
    {

        MiRunnable miRunnable1 = new MiRunnable(1000);
        MiRunnable miRunnable2 = new MiRunnable(2000);

        FutureTask<String>
            futureTask1 = new FutureTask<>(miRunnable1,
                                           "FutureTask1 is complete");
        FutureTask<String>
            futureTask2 = new FutureTask<>(miRunnable2,
                                           "FutureTask2 is complete");

        // create thread pool of 2 size for ExecutorService
        ExecutorService executor = Executors.newFixedThreadPool(2);

        // submit futureTask1 to ExecutorService
        executor.submit(futureTask1);

        // submit futureTask2 to ExecutorService
        executor.submit(futureTask2);

        while (true) {
            try {
```

```
                if (futureTask1.isDone() && futureTask2.isDone()) {

                    System.out.println("Las dos tareas completadas");

                    // shut down executor service
                    executor.shutdown();
                    return;
                }

                if (!futureTask1.isDone()) {


                    System.out.println("Salida de FutureTask1  = "
                                       + futureTask1.get());
                }

                System.out.println("Esperado a FutureTask2 ");

                //
                String s = futureTask2.get(250, TimeUnit.MILLISECONDS);

                if (s != null) {
                    System.out.println("FutureTask2 salida=" + s);
                }
            }

            catch (Exception e) {
                System.out.println("Exception: " + e);
            }
        }
    }
}
```

Nota: Recordar que con **FutureTask** también **podemos ejecutar objetos de tipo Callable**. Lo vais a hacer en los siguientes ejercicios.

10 Ejercicios

1. Modificar **TareaFutureEsPrimo** para **lanzar la tarea Callable** con un FutureTask. Llamarla TareaEsPrimoFutureTask.
2. Modificar el ejemplo anterior para lanzar el objeto Runnable con un Future directamente con el Executor. Llamarla **EjemploFutureRunnable.**