

Tema 3. Ampliación de contenidos del libro

1	Introducción.....	2
2	Patrones de arquitectura de servidor.....	2
2.1	Patrón en capas	3
2.2	Patrón cliente servidor	4
2.3	Master-slave pattern.....	5
2.4	Patrón filtro tubería.	6
2.5	Broker pattern	6
2.6	Patrón Peer-to-peer	8
2.7	Patrón de bus de eventos.....	9
2.8	Model-view-controller pattern	10
2.9	Patrón Blackboard.....	12
2.10	Patrón interprete.....	13
3	Fork/Join Framework	14
3.1	Patron fork-join.....	14
3.1.1	Fork.....	14
3.1.2	Join	15
3.2	Modelo de trabajo	16
3.3	Algoritmo de robo de trabajo	17
3.4	Instanciación de ForkJoinPool.....	17
3.5	ForkJoinTask<V>	18
3.6	Otra manera no recursiva de aprovechar el paralelismo y java 8.	22
3.7	Indicaciones	26
3.8	Paralelizando hilos con el modelo fork join	27
3.9	Uso de callables	29
3.10	Usando el fork join pool con Callables.....	38
3.11	Ejemplo con paralelismo hilos y Streams.....	41
3.12	Comparativa de velocidades de Streams paralelos y secuenciales. Obligatorio probarlo.....	43

4	Completable Futures.....	45
5	Modelos de diseño de servidores.....	55
5.1	La BlockingQueue de java.....	55
5.2	La priority blocking queue.....	60
5.3	Servidor Active Object	64
5.4	Ejercicio	81
5.5	Modelo de servidor objeto Stub.....	82
5.6	Nuestro modelo.....	83
6	Servidor de ficheros. Tarea.....	86

1 Introducción

En el tema 2 **introducimos el modelo de Futures y Callables de java**. Vamos a seguir **avanzando en las diferentes versiones de java hasta llegar a java 9**, teniendo en cuenta los nuevos modelos de ejecución de tareas, **ForkJoinFramework** y **CompletableFuture**.

Empezaremos con el **ForkJoinFramework**. En el tema anterior usabamos un **pool de hilos concurrente**, **basicamente** usaban un procesador para realizar la ejecución. En **nuestros nuevos servidores y ordenadores con multiples nucleos** nos tenemos que mover **hacia un pool paralelo**. Esto es lo que nos ofrece el **framework ForkJoin**, que nuestros hilos puedan ejecutarse en diferentes procesadores. **Igualmente veremos en este tema las CompletableFuture**, que **apoyandose** en el interfaz future permiten **operaciones asincronas** y se ejecutan **si no se define ningun pool de hilos** en el pool de hilos ForkJoin, **introduciendo además paralelismo en nuestras Futures**.

Recordar que 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.

Pero empezaremos por los patrones de arquitectura para servidor más comunes.

2 Patrones de arquitectura de servidor

¿Qué es un patrón arquitectónico?

Según Wikipedia,

Un patrón arquitectónico es una solución general y reutilizable a un problema que ocurre comúnmente en la arquitectura de software dentro de un contexto determinado. Los patrones arquitectónicos son similares al patrón de diseño de software, pero tienen un alcance más amplio.

En este tema, se **va a explicar brevemente los siguientes 10 patrones** arquitectónicos comunes con su uso, pros y contras.

1. **Layered pattern, patron en capas.**
2. **Client-server pattern, patron cliente servidor.**
3. **Master-slave pattern, patron maestro esclavo**
4. **Pipe-filter pattern, patron filtro tubería.**
5. **Broker pattern, patron broker.**
6. **Peer-to-peer pattern, patron P2P.**
7. **Event-bus pattern. Patrón bus evento.**
8. **Model-view-controller pattern. Patrón MVC.**
9. **Blackboard pattern. Patrón blackboard.**
10. **Interpreter pattern. Patron interpreter**

2.1 Patrón en capas

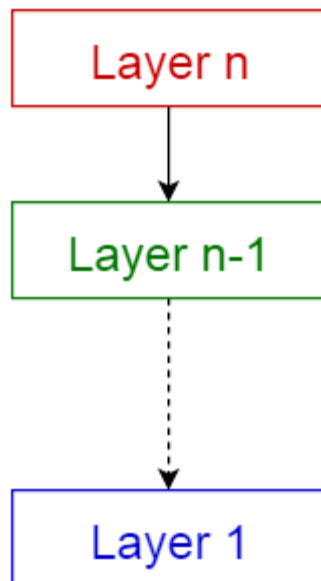
Este patrón se puede utilizar para estructurar programas que se pueden descomponer en grupos de subtarear, cada una de las cuales se encuentra en un nivel particular de abstracción. Cada capa proporciona servicios a la siguiente capa superior.

Las 4 capas más comúnmente encontradas de un sistema de información general son las siguientes.

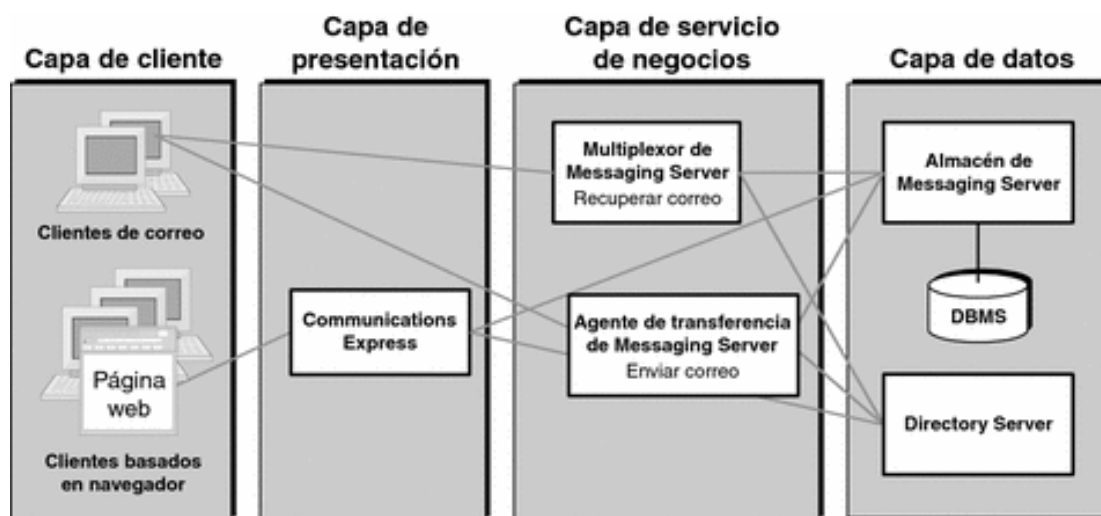
- **Capa de presentación** (también conocida como **capa de interfaz de usuario**)
- **Capa de aplicación** (también conocida como **capa de servicio**)
- **Capa de lógica empresarial** (también conocida como **capa de dominio**)
- **Capa de acceso a datos** (también conocida como **capa de persistencia**)

Uso

- Aplicaciones de escritorio.
- Aplicaciones web de comercio electrónico.



Ejemplo de aplicación en tres capas

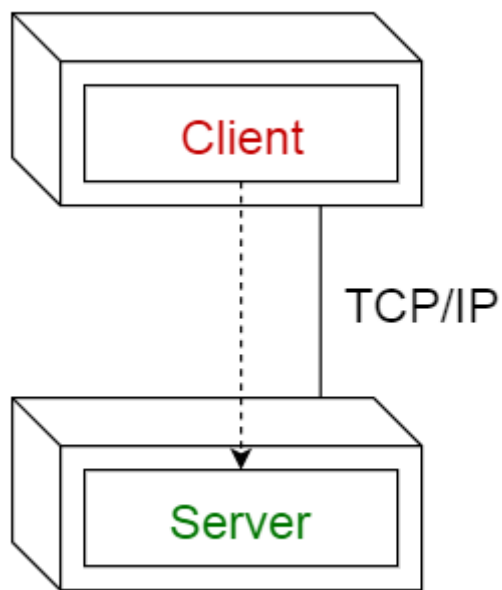


2.2 Patrón cliente servidor

Este patrón consta de dos partes; un **servidor** y varios **clientes**. El **componente de servidor** **proporcionará servicios a varios componentes de cliente**. Los clientes **solicitan servicios del servidor** y el **servidor proporciona servicios relevantes** a esos clientes. Además, el **servidor sigue escuchando las solicitudes de cliente**.

Uso

Aplicaciones en línea como correo electrónico, intercambio de documentos y banca.

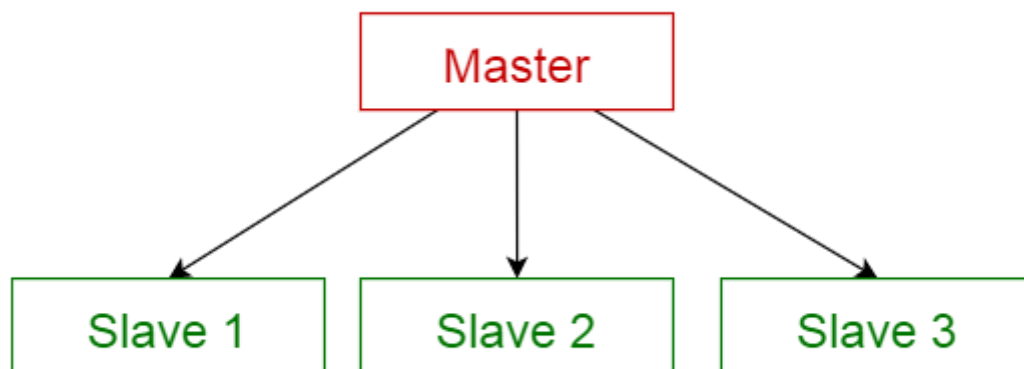


2.3 Master-slave pattern

Este patrón consta de dos partes; maestro y esclavos. El componente maestro distribuye el trabajo entre componentes esclavos idénticos y calcula un resultado o resultados finales a partir de los resultados que devuelven los esclavos.

Uso

- En la **replicación de base de datos**, la **base de datos maestra** se considera como el **origen autorizado** y las bases de datos esclavas se sincronizan con ella.
- **Periféricos conectados a un bus** en un **sistema informático (unidades maestras y esclavas)**.

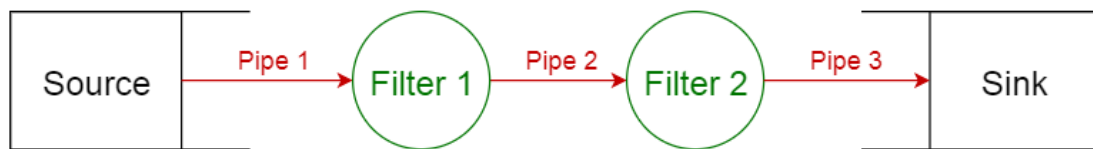


2.4 Patrón filtro tubería.

Este **patrón se puede utilizar para estructurar sistemas que producen y procesan un flujo de datos**. Cada **paso de procesamiento se incluye dentro de un componente de filtro**. Los **datos a procesar se pasan a través de tuberías**. Estas **canalizaciones se pueden utilizar para el almacenamiento en búfer o para fines de sincronización**.

Uso

- Compiladores. Los **filtros consecutivos realizan análisis léxicos, análisis, análisis semántico y generación de código**.
- Flujos de **trabajo en bioinformática**.



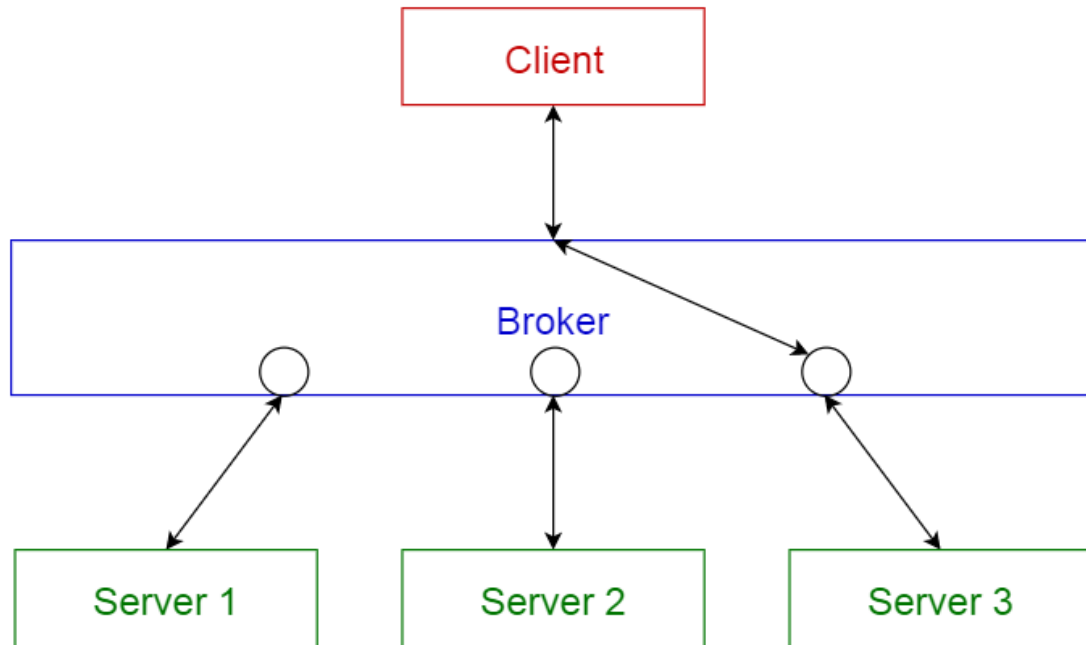
2.5 Broker pattern

Este **patrón se utiliza para estructurar sistemas distribuidos con componentes desacoplados**. Estos **componentes pueden interactuar entre sí mediante invocaciones de servicio remoto**. Un **componente de intermediario es responsable de la coordinación de la comunicación entre los componentes**.

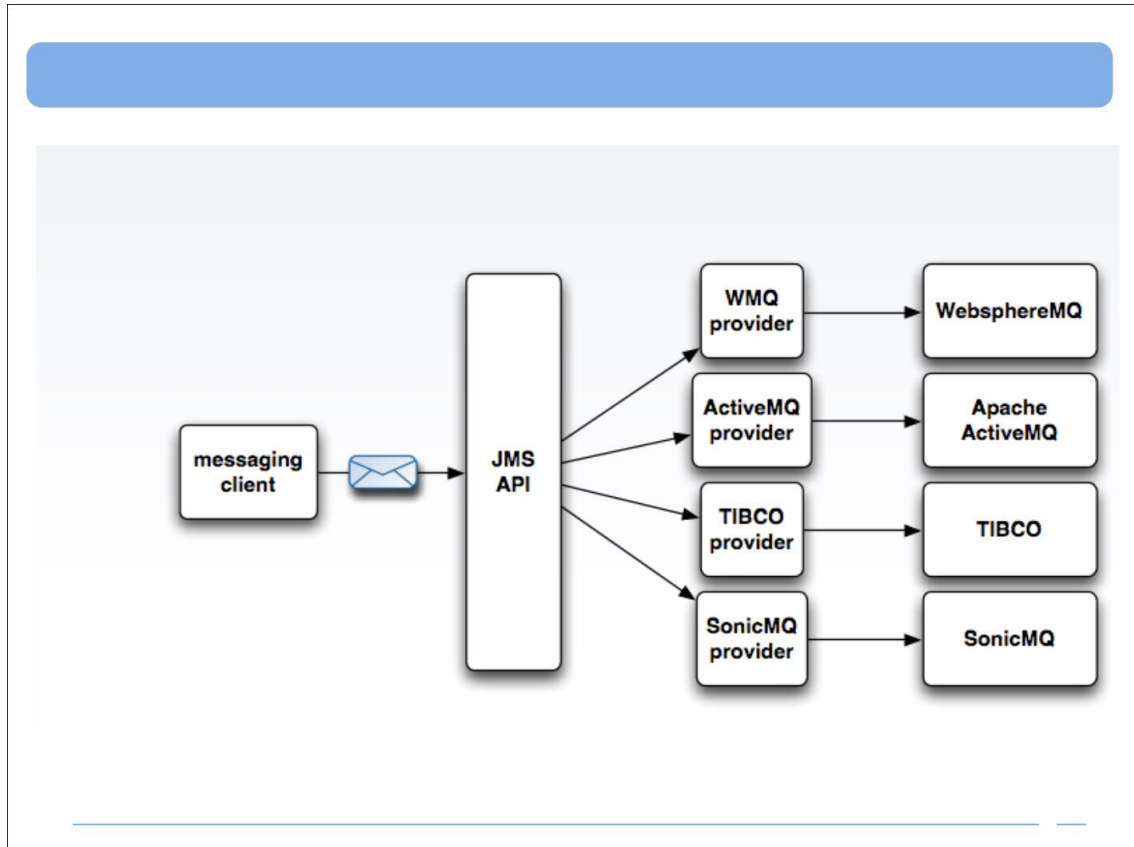
Los **servidores publican sus capacidades (servicios y características) en un intermediario**. Los **clientes solicitan un servicio al intermediario y, a continuación, el intermediario redirige al cliente a un servicio adecuado desde su registro**.

Uso

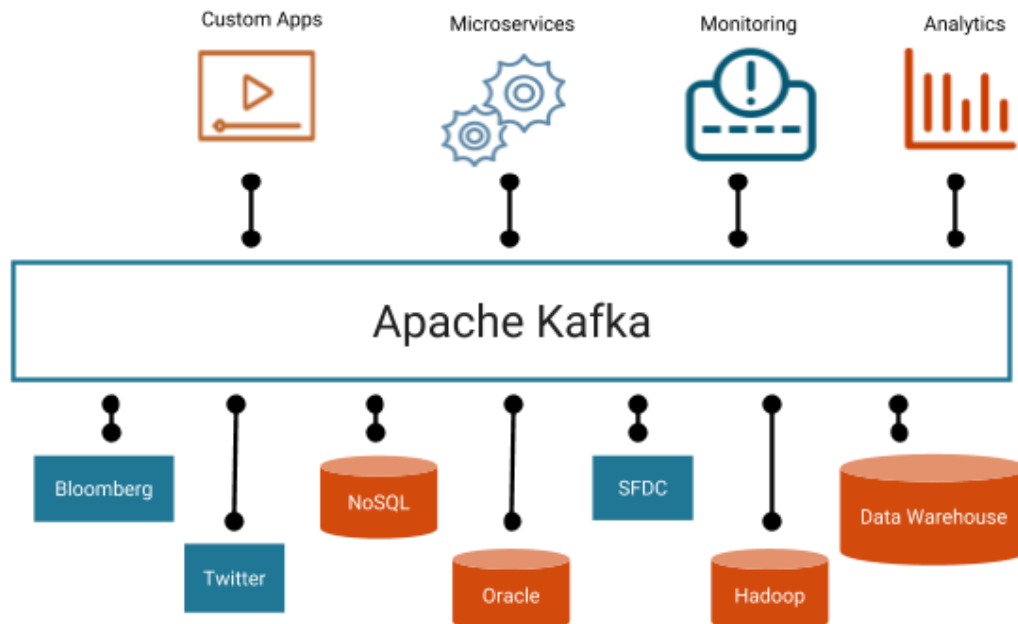
Software de **agente de mensajes como Apache ActiveMQ, Apache Kafka, RabbitMQ y JBoss Messaging**.



Un ejemplo real sería una API de mensajería con su Broker que elige entre diferentes proveedores y manda la petición al correcto.



Otro ejemplo, podría ser **una aplicación con el Servidor Apache Kafka** que redirige a **diferentes servicios, base de datos, relacional o no relacional, redes sociales, a diferentes clientes**, cliente de **escritorio o web**, cliente de **analítica**, cliente de **microservicios**, dependiendo de **la petición del cliente y el tipo de cliente**.

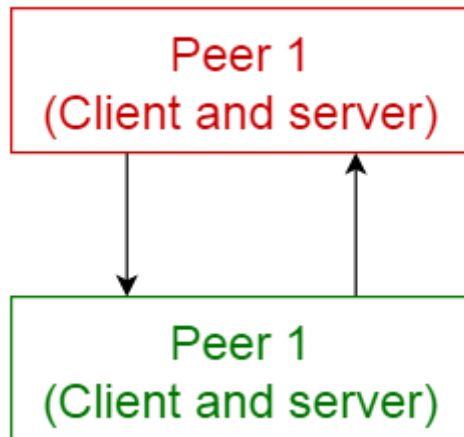


2.6 Patrón Peer-to-peer

En este patrón, **los componentes individuales se conocen como pares**. Los **pares pueden funcionar tanto como un cliente, solicitando servicios** de otros pares, como **como un servidor, proporcionando servicios** a otros pares. Un par puede actuar como cliente o como servidor o como ambos, y puede cambiar su rol dinámicamente con el tiempo.

Uso

- Redes de **intercambio de archivos** como Gnutella y G2)
- **Protocolos multimedia** como P2PTV y PDTP.
- Productos basados en **criptomonedas** como Bitcoin y Blockchain
- **Emule**

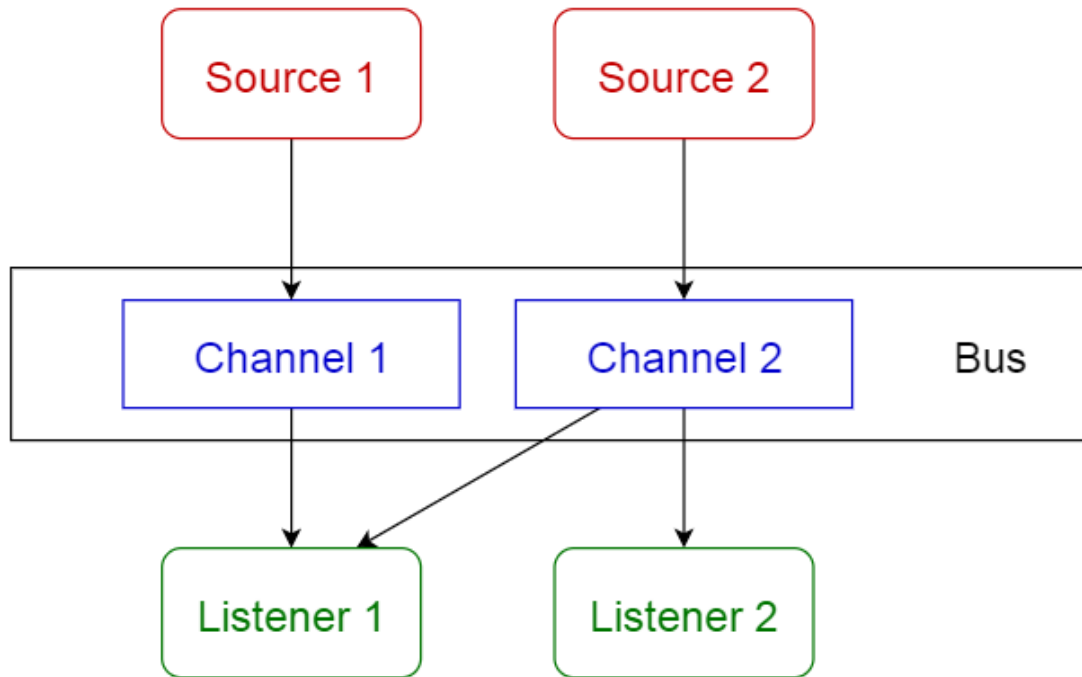


2.7 Patrón de bus de eventos

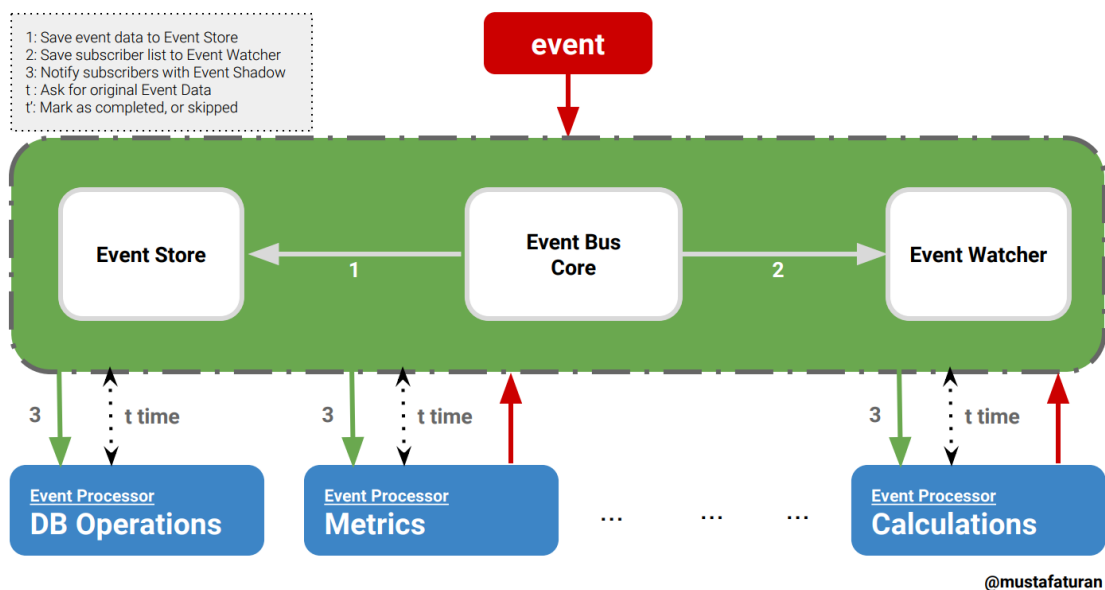
Este patrón se ocupa principalmente de los eventos y tiene 4 componentes principales; **origen del evento**, **detector de eventos**, **canal** y **bus de eventos**. Los **orígenes publican mensajes en determinados canales** en un **bus de eventos**. Los **oyentes se suscriben a canales particulares**. Los agentes de escucha reciben notificaciones de los mensajes que se publican en un canal al que se han suscrito antes. Se relaciona con el **patrón de diseño observer**.

Uso

- Desarrollo de Android
- Servicios de notificación



Patron de eventos para firebase de Android.



2.8 Model-view-controller pattern. Modelo Vista Controlador

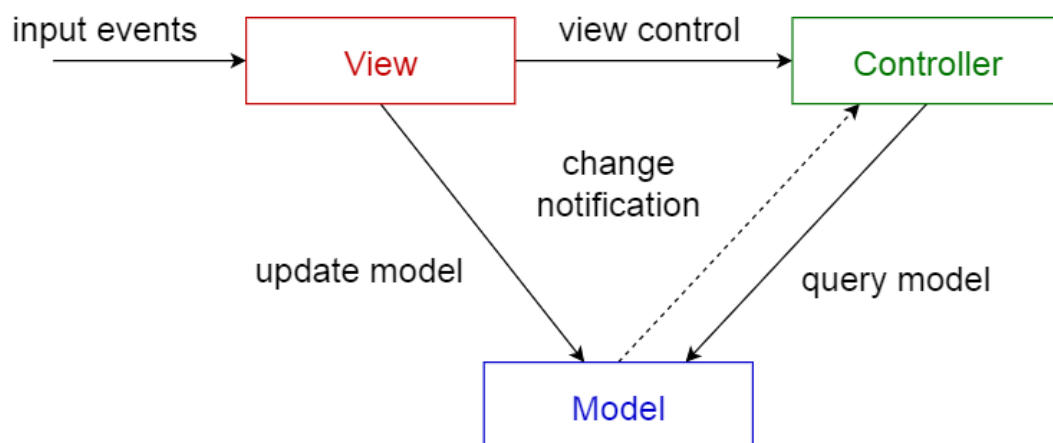
Este patrón, también conocido como **patrón MVC**, divide una aplicación interactiva en 3 partes como,

- **modelo:** contiene la funcionalidad principal y los datos
- **vista:** muestra la información al usuario (se puede definir más de una vista)
- **controlador:** maneja la entrada del usuario

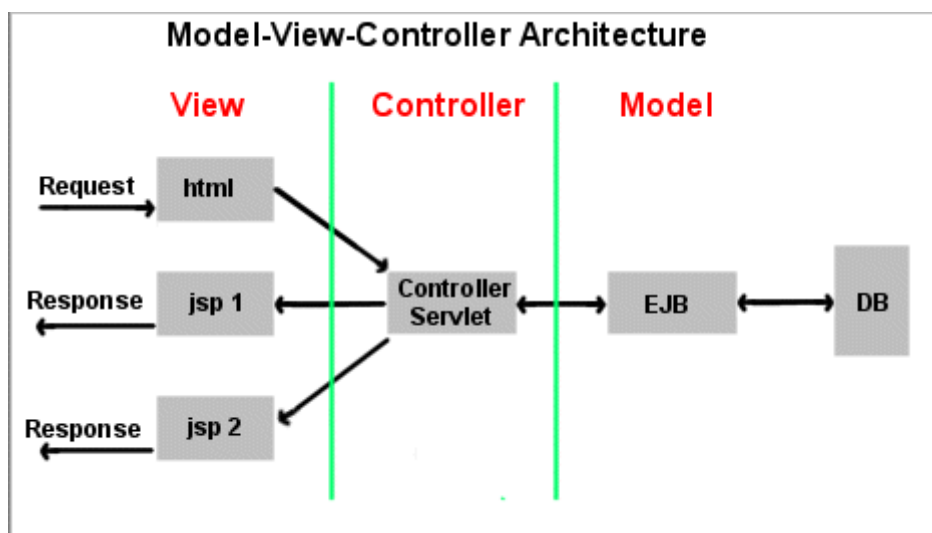
Esto se hace **para separar las representaciones internas de la información** de las formas en que la información se y se acepta desde el usuario. Desacopla los componentes y **permite una reutilización eficiente del código**.

Uso

- Arquitectura **para aplicaciones World Wide Web** en los principales lenguajes de programación.
- Marcos web como **Django** y **Rails**.



En acceso a datos realizareis un patrón similar a este



2.9 Patrón Blackboard

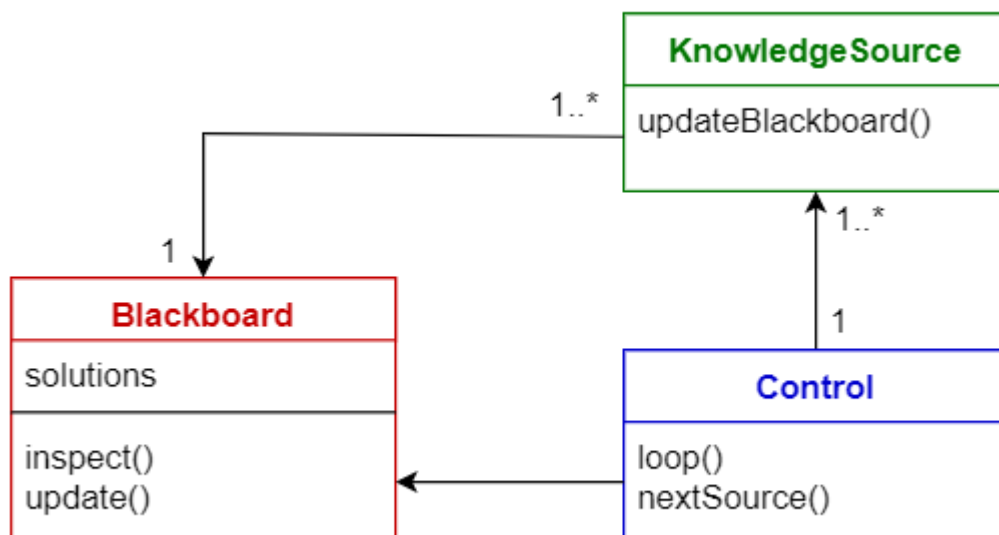
Este **patrón es útil para problemas** para los que **no se conocen estrategias de solución** deterministas. El patrón de pizarra consta de 3 componentes principales.

- **Pizarra** — una memoria global estructurada que contiene objetos del espacio de solución
- **Fuente de conocimiento:** módulos especializados con su propia representación
- **Componente de control:** selecciona, configura y ejecuta módulos.

Todos los **componentes tienen acceso a la pizarra**. Los componentes **pueden producir nuevos objetos de datos que se agregan a la pizarra**, para aportar nuevas soluciones a los problemas. Los **componentes buscan determinados tipos de datos en la pizarra y pueden encontrarlos por coincidencia de patrones con la fuente de conocimiento** existente. Es muy utilizado en **inteligencia artificial**.

Uso

- Reconocimiento de voz
- Identificación y seguimiento del vehículo
- Identificación de la estructura proteica
- Interpretación de señales de sonar.

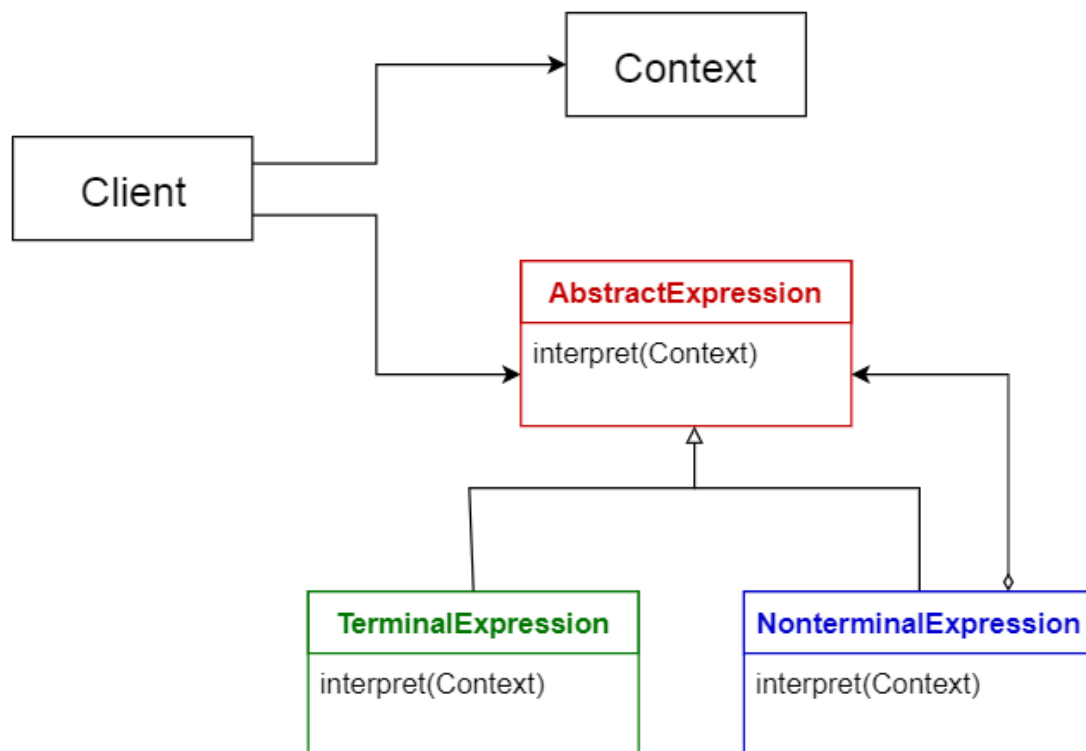


2.10 Patrón interprete

Este patrón se utiliza para diseñar un **componente que interpreta programas escritos en un lenguaje dedicado**. Especifica principalmente cómo **evaluar líneas de programas, conocidas como oraciones o expresiones escritas** en un idioma determinado. La idea básica es tener una clase para cada símbolo del lenguaje.

Uso

- Lenguajes de consulta de base de datos como SQL.
- Idiomas utilizados para describir protocolos de comunicación.
- Interpretación del lenguaje natural.



3 Fork/Join Framework. Paralelismo en Java

El **fork/join framework** es una implementación de la interfaz `ExecutorService` que te ayuda a aprovechar los múltiples procesadores. Está **diseñado para trabajos que se pueden dividir en piezas más pequeñas de forma recursiva**. El objetivo es **utilizar toda la potencia de procesamiento disponible** para mejorar el rendimiento de la aplicación.

Al igual que con cualquier **implementación de `ExecutorService`**, el **fork/join Framework** distribuye tareas a hilos de trabajo o `workerthreads` en un grupo de hilos. El **framework fork-join** es distinto porque utiliza un **algoritmo de robo de trabajo**. Los hilos de trabajo, `workerthreads` que se quedan sin tareas en su cola que hacer pueden “roban” tareas de otros `workerthreads` que todavía están ocupados.

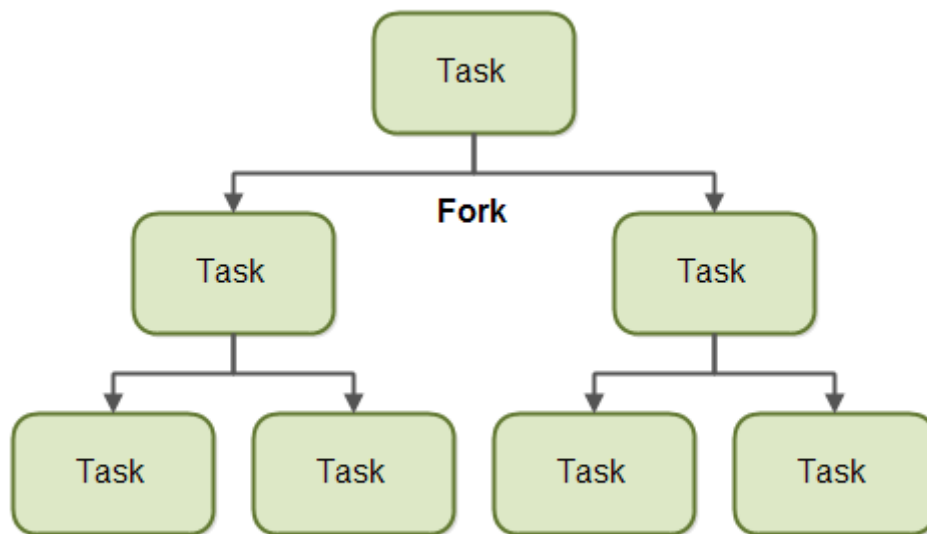
El punto central **del fork/join framework es la clase `ForkJoinPool`**, esta clase hereda de **`AbstractExecutorService`**. **`ForkJoinPool` implementa el algoritmo de robo de trabajo principal y puede ejecutar procesos `ForkJoinTask`**.

3.1 Patron fork-join

Antes de estudiar el `ForkJoinPool` vamos a **explicar cómo el modelo fork-join en general funciona**. El principio fork-join (de **bifurcación y combinación**) consta de dos pasos que **se realizan de forma recursiva**. Estos dos pasos son el **paso de la bifurcación, fork** y el **paso de unión, join**.

3.1.1 Fork

Una **tarea que utiliza el principio fork-join puede *bifurcarse* (dividirse) en subtareas más pequeñas que se pueden ejecutar simultáneamente**. Se ilustra en el siguiente diagrama:



Al **dividirse en subtareas**, cada subtarea se puede **ejecutar en paralelo** con diferentes CPU o hilos diferentes en la misma CPU.

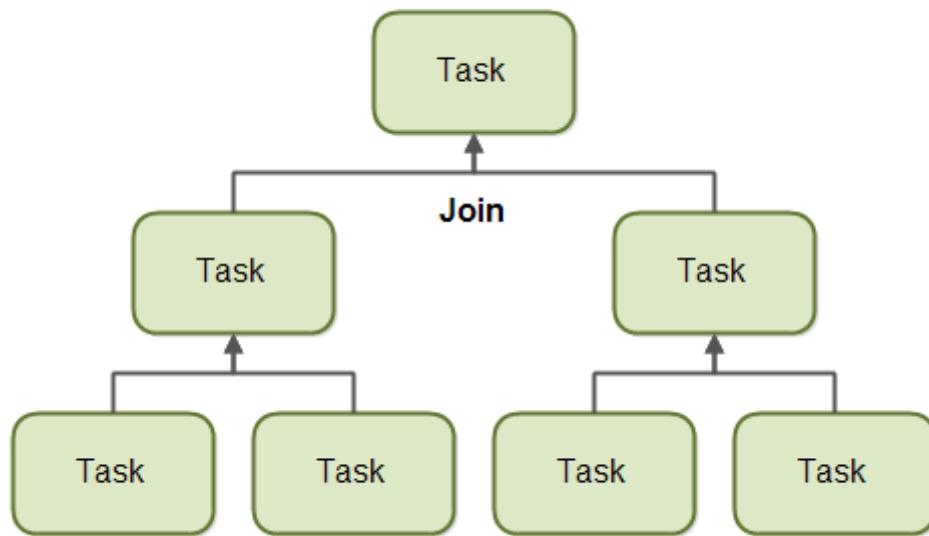
Una tarea solo se **divide en subtareas** si el **trabajo que se le dio a la tarea** es lo **suficientemente grande** como para que esto tenga sentido. Hay una **sobrecarga de ejecución** para dividir una tarea en subtareas, por lo que para **pequeñas cantidades de trabajo** esta sobrecarga puede ser mayor que la **aceleración lograda** mediante la ejecución simultánea de subtareas.

El **límite para cuando tiene sentido bifurcar** una tarea en subtareas **también se denomina umbral o threshold**. Depende de cada tarea decidir un umbral sensato. Depende en gran medida del tipo de trabajo que se está haciendo.

3.1.2 Join

Cuando **una tarea se ha dividido en subtareas**, la **tarea espera hasta que las subtareas hayan terminado** de ejecutarse.

Una vez que **las subtareas han terminado de ejecutarse**, la **tarea puede unir (combinar) todos los resultados** en un resultado. Se ilustra en el siguiente diagrama:



Por supuesto, no todos los tipos de tareas deben devolver un resultado. Si **las tareas no devuelven un resultado**, una tarea solo espera a que se completen sus subtareas. No se produce ninguna fusión de resultados.

3.2 Modelo de trabajo

El **modelo de framework fork/join** fue presentado en Java 7. Provee de herramientas para **acelerar el procesamiento paralelo**, intentando usar todos los núcleos de tu procesador con la estrategia **divide y vencerás**.

En la práctica, esto significa que el framework en la parte “fork” primero **divide la tarea en tareas más pequeñas independientes**, hasta que hay son suficientemente simples para ser ejecutadas de manera asíncrona.

Después **comienza la parte join**, todas las pequeñas tareas se **unen recursivamente para ofrecer un único resultado**. Si las tareas no devuelven nada (void) el programa simplemente espera a que cada subtask sea ejecutada.

Para **proporcionar una ejecución paralela eficaz**, el marco de trabajo de forkJoin utiliza un grupo de pool de hilos denominado **ForkJoinPool**, que administra hilos de trabajo de tipo **ForkJoinWorkerThread**.

3.3 Algoritmo de robo de trabajo

En otras palabras, los hilos liberados intentan robar el trabajo de los hilos en cola u ocupados.

La idea es que tenemos unos hilos predefinidos llamados **workerthread** en el framework que se reparten paralelamente entre procesadores. Cada **hilo tiene una cola donde se van introduciendo los hilos o tareas que crean los programadores** que usan el framework. Cada **workerthread del hilo extrae uno de nuestros hilos o tareas y lo ejecuta**. Cuando termina cogen otro de la cabeza o principio de la cola. Si su cola esta vacia coge tareas o hilos de otra cola del workerthread más ocupado, porque seguramente ese workerthread que esta más ocupado, es por que los hilos o tareas que ha recibido son más pesadas.

Este enfoque minimiza la posibilidad de que los workerthreads compitan por las tareas. También reduce el número de veces que el hilo tendrá que ir en busca de trabajo, ya que su objetivo son trabajo más pesados disponibles primero.

3.4 Instanciación de ForkJoinPool

En Java 8, 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);
```

3.5 ForkJoinTask<V>

ForkJoinTask es la clase base para las tareas ejecutadas dentro de **ForkJoinPool**. En la práctica, **se debe utilizar una de sus dos subclases: RecursiveAction** para las tareas que **no devuelven resultado** y **RecursiveTask<V>** para las tareas que **devuelven un valor**. Ambos tienen un método abstracto `compute()` en el que se define la lógica de la tarea.

Vamos a **ver un ejemplo con RecursiveTask** de como usar estas tareas y el **paralelismo**. Con un ejemplo sobraré para que **entendáis como repartir el trabajo de manera paralela** con este framework

En esta tarea recursiva vamos a **dividir un array de 1000 numeros**, en problemas más **pequeños de 10**. Es decir, vamos a lanzar un pequeño **hilo o tarea por cada 10 numeros del array de manera recursiva**. Si la **diferencia entre máximo y mínimo es menor o igual que 10** hacemos la suma de los diez números. Sino, **dividimos el problema por la mitad**. Esto se conoce **en lenguaje técnico como threshold, umbral** o caso base para esta tarea recursiva, en **este caso 10**.

Caso base o umbral:

```
if(maximo - minimo <= 10) {  
    long sum = 0;  
  
    for(int i = minimo; i < maximo; ++i)  
        sum += array[i];  
    return sum;  
}
```

Sino **dividimos el problema recursivamente, a la mitad**. La primera vez será 500 y 500 para izq y der.

```
int med = minimo + (maximo - minimo) / 2;
    Sum izq = new Sum(array, minimo, med);
    Sum der = new Sum(array, med, maximo);
    izq.fork();
    long resultadoDerecha = der.compute();
    long resultadoIzquierda = izq.join();
    return resultadoIzquierda + resultadoDerecha;
```

Fijaos como hacemos un `fork()` de la izquierda, para que lance la tarea `Sum(array,0,500)` la primera vez, en un **threadworker** aparte.

En nuestro **threadWorker** actual hacemos el `compute` de la parte derecha `Sum(array, 500,1000)`.

Teniendo en cuenta que el umbral es diez, el total de números mil, dividiendo, hacemos 50 forks, y 50 computes, 50 derechas y 50 izquierdas, lanzamos en total 100 tareas, que serán resueltas por tantos threadworkers como procesadores tengamos, en mi caso 12. Las 50 tareas de la izquierda con el fork son asignadas a las colas de nuevos threadworkers. Las 50 de la derecha, al threadworker inicial porque hacemos compute. Pero con el algoritmo de robo de trabajo, se repartirán correctamente, posteriormente.

Las tareas izquierdas esperan a las derechas con el join. Se unen todas al final y se junta el resultado en `return resultadoIzquierda + resultadoDerecha;`

En la función `main` de este programa podéis ver como obtenemos el máximo de procesadores disponibles

```
int nThreadWorkers = Runtime.getRuntime().availableProcessors();
```

Creamos el pool de hilos threadworkers que ejecutará nuestras tareas. En mi caso es 12. Tendremos 12 threadworkers para ejecutar 100 tareas.

```
ForkJoinPool forkJoinPool = new ForkJoinPool(nThreadsWorkers);
```

Lanzamos la ejecución inicial

```
Long resultado = forkJoinPool.invoke(new Sum(numeros,0,numeros.length));
```

```
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ForkJoinPool;
import java.util.concurrent.RecursiveTask;

public class RecursiveParalelo {

    static class Sum extends RecursiveTask<Long> {
        int minimo;
        int maximo;
        int[] array;

        Sum(int[] array, int minimo, int maximo) {
            this.array = array;
            this.minimo = minimo;
            this.maximo = maximo;
        }

        protected Long compute() {

            if(maximo - minimo <= 10) {
                long sum = 0;

                for(int i = minimo; i < maximo; ++i)
                    sum += array[i];
                return sum;
            } else {
```

```
        int med = minimo + (maximo - minimo) / 2;
        Sum izq = new Sum(array, minimo, med);
        Sum der = new Sum(array, med, maximo);
        izq.fork();
        long resultadoDerecha = der.compute();
        long resultadoIzquierda = izq.join();
        return resultadoIzquierda + resultadoDerecha;
    }
}

}

}

public static void main(final String[] arguments) throws InterruptedException,
    ExecutionException {

    int nThreadsWorkers = Runtime.getRuntime().availableProcessors();
    System.out.println(nThreadsWorkers);

    int[] numeros = new int[1000];

    for(int i = 0; i < numeros.length; i++) {
        numeros[i] = i;
    }

    ForkJoinPool forkJoinPool = new ForkJoinPool(nThreadsWorkers);
    Long resultado = forkJoinPool.invoke(new Sum(numeros,0,numeros.length));
    System.out.println(resultado);
}
}
```

3.6 Otra manera no recursiva de aprovechar el paralelismo y java 8.

Aunque en principio **el modelo fork join** esta diseñado para realizar el **paralelismo recursivo**, podemos conseguir **paralelismo lanzando varias tareas dentro de una tarea RecursiveAction or RecursiveTask**. En el siguiente ejemplo invocamos a la TareaParalela desde el pool de hilos:

```
forkJoinPool.invoke(new TareaParalela("Tarea 1"));
```

Después lanzamos dentro del compute de TareaParalela, otras dos tareas con un fork. De esta manera, **teoricamente podemos conseguir ejecutar tareas en procesadores distintos**.

```
TareaParalela2 ta2= new TareaParalela2("Tarea2");
TareaParalela3 ta3= new TareaParalela3("Tarea3");

        ta3.fork();
        ta2.fork();
```

ForkParalelo.java

```
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ForkJoinPool;
import java.util.concurrent.RecursiveAction;

public class ForkParalelo {

    static class TareaParalela extends RecursiveAction{

        String nombreTarea="";
```

```
public TareaParalela(String nombreTarea ) {

    this.nombreTarea=nombreTarea;

}

@Override
protected void compute() {
    // TODO Auto-generated method stub

    TareaParalela2 ta2= new TareaParalela2("Tarea2");
    TareaParalela3 ta3= new TareaParalela3("Tarea3");

    ta3.fork();
    ta2.fork();

    subtarea1();

    ta2.join();
    ta3.join();

}

private void subtarea1 () {

    try {

        Thread.sleep(5000);
    } catch (InterruptedException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    }

}
```

```
        System.out.println("sub Tarea1 " + nombreTarea + " termina ")
;

    }

}

static class TareaParalela2 extends RecursiveAction {

    String nombreTarea="";

    public TareaParalela2(String nombreTarea ) {

        this.nombreTarea=nombreTarea;

    }

    @Override
    protected void compute() {
        // TODO Auto-generated method stub

        try {

            Thread.sleep(2000);
        } catch (InterruptedException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }

        System.out.println("Tarea2 " + nombreTarea + " termina ");
    }
}
```



```
    }

}

static class TareaParalela3 extends RecursiveAction {

    String nombreTarea="";

    public TareaParalela3(String nombreTarea ) {

        this.nombreTarea=nombreTarea;

    }

    @Override
    protected void compute() {
        // TODO Auto-generated method stub

        try {

            Thread.sleep(3000);
        } catch (InterruptedException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }

        System.out.println("Tarea3 " + nombreTarea + " termina ");
    }
}
```

```
    }

    }

    public static void main(final String[] arguments) throws InterruptedException,
        ExecutionException {

        int nThreadsWorkers = Runtime.getRuntime().availableProcessors();
        System.out.println(nThreadsWorkers);

        ForkJoinPool forkJoinPool = new ForkJoinPool(nThreadsWorkers);
        forkJoinPool.invoke(new TareaParalela("Tarea 1"));

    }

}
```

3.7 Indicaciones

Usado el **fork/join framework** se puede **acelerar el procesamiento** de tareas pesadas, pero para lograr este resultado, se deben seguir algunas directrices:

1. Es **mejor usar el menor número posible de grupos de threadworkers**: en la mayoría de los casos, **la mejor decisión es usar un grupo de hilos por aplicación o sistema**.
2. Igualmente **es mejor usar el common pool si no se necesita especificar el número de procesadores a usar**.

3. **Utilizar un umbral razonable para dividir ForkJoinTask** en subtareas
4. **Evitar cualquier bloqueo** en tus tareas ForkJoin

3.8 Paralelizando hilos con el modelo fork join

En el siguiente ejemplo **vamos a usar el pool de hilos fork join** para paralelizar la **ejecución de nuestros hilos en múltiples procesadores**.

Lo primero creamos un pool de hilos fork join

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

Lo **segundo igual que hacíamos en el tema anterior con ExecuteService**, pero esta vez **paralelo hacemos un submit de los hilos que les hace ejecutarse** como tareas ForkJoinTasks .

```
ForkJoinTask<?> tareaHilo1 = forkJoinPool.submit(hilo1);  
ForkJoinTask<?> tareaHilo2 = forkJoinPool.submit(hilo2);
```

Finalmente esperamos a que los hilos esperen el uno al otro con join.

```
tareaHilo1.join();  
tareaHilo2.join();
```

```
import java.util.concurrent.ExecutionException;  
import java.util.concurrent.ForkJoinPool;
```

```
import java.util.concurrent.ForkJoinTask;

import pooldehilosparalelo.ForkParalelo.TareaParalela;

public class ForkConHilosParalelos {

    public static void main(final String[] arguments) throws InterruptedException,
        ExecutionException {

        ForkJoinPool forkJoinPool = new ForkJoinPool(2);

        Thread hilo1 = new Thread() {

            public void run() {

                try {

                    Thread.sleep(5000);

                    System.out.println("Hilo 1 terminado");
                } catch (InterruptedException e) {
                    // TODO Auto-generated catch block
                    e.printStackTrace();
                }
            }

        };

        Thread hilo2 = new Thread() {
```

```
        public void run() {

            try {
                Thread.sleep(1000);
                System.out.println("Hilo 2 terminado");
            } catch (InterruptedException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }

        }

    };

    ForkJoinTask<?> tareaHilo1 = forkJoinPool.submit(hilo1);
    ForkJoinTask<?> tareaHilo2 = forkJoinPool.submit(hilo2);

    tareaHilo1.join();

    tareaHilo2.join();

}

}
```

3.9 Uso de callables

Debería haberse indicado en el tema anterior pero para clarificar el uso de callables en entorno cliente vamos a añadir este ejemplo. En este programa tenemos una ventana sobre la que introducimos un número calculamos si es primo con una callable y una **tarea future**, y calculamos si es primo. En la callable **hemos introducido un retraso de 5**

segundos para simular la respuesta de un servidor cuando no tenemos buena conexión.

Vamos a **analizar lo que ocurre en el evento onclick del botón** que es **donde transcurre la lógica de nuestro programa**. Empezamos **cambiando el texto de la etiqueta donde escribiremos el resultado final** a esperando resultado. Por la naturaleza **síncrona de las Futures puede que no funcione**. Para eso introduciremos después las **CompletableFutures que proporcionan un ejecución asíncrona**. Cogemos igualmente el valor numérico del campo de texto

```
lblNewLabel_2.setText("Esperando resultado ");  
int num = Integer.valueOf(textField.getText());
```

Hacemos **un submit de la future task que ejecutará la callable**. Lo realizamos con el nuevo pool ForkJoinPool. **Desde el primer momento que hacemos el submit la tarea empieza a ejecutarse**. Esperamos con el **while a que la tarea se complete indicando en la consola que estamos esperando un resultado**.

```
Future<Boolean>tareaCallable1 = ForkJoinPool.commonPool().submit(new PrimoCal  
lable2(num) );  
  
        while (!tareaCallable1.isDone() && !tareaCallable1.isCancelled()  
) {  
  
                System.out.println("Esperando Resultado");  
        }
```

Cuando **termina la tarea recogemos el resultado con el get y los mostramos en la etiqueta que esta oculta**.

```
if (tareaCallable1.get()) {  
        lblNewLabel_2.setText("El numero es primo");  
}
```

```
        } else {

            lblNewLabel_2.setText("El numero no es primo");

        }
    } catch (InterruptedException e1) {
        // TODO Auto-generated catch block
        e1.printStackTrace();
    } catch (ExecutionException e1) {
        // TODO Auto-generated catch block
        e1.printStackTrace();
    }
}
```

Evento onClick del boton

```
public void actionPerformed(ActionEvent e) {

    lblNewLabel_2.setText("Esperando resultado ");

    int num = Integer.valueOf(textField.getText());

    Future<Boolean>tareaCallable1 = ForkJoinPool.commonPool().
    submit(new PrimoCallable2(num) );

    while (!tareaCallable1.isDone() && !tareaCallable1.isCanceled()) {

        System.out.println("Esperando Resultado");

    }

    try {
        if (tareaCallable1.get()) {
```

```
        lblNewLabel_2.setText("El numero es primo");

        } else {

            lblNewLabel_2.setText("El numero no es primo");
        }
    } catch (InterruptedException e1) {
        // TODO Auto-generated catch block
        e1.printStackTrace();
    } catch (ExecutionException e1) {
        // TODO Auto-generated catch block
        e1.printStackTrace();
    }

}

});
```

VentanaCallable.java

```
import java.awt.EventQueue;
import javax.swing.JFrame;
import javax.swing.JLabel;
import java.awt.Font;
import javax.swing.JTextField;
import javax.swing.JButton;
```



```
import java.awt.event.ActionListener;
import java.util.concurrent.Callable;
import java.util.concurrent.CompletableFuture;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ForkJoinPool;
import java.util.concurrent.Future;
import java.util.concurrent.TimeUnit;
import java.awt.event.ActionEvent;

class PrimoCallable2 implements Callable {
    private int numero;

    public PrimoCallable2(int numero) {
        this.numero = numero;
    }

    private static boolean esPrimo(int n)
    {
        boolean continuar = true;
        boolean esPrimo = true;
        long divisor = 2;

        try {
            Thread.sleep(5000);
        } catch (InterruptedException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
        do {
            if (n % divisor == 0) {
                continuar = false;
                esPrimo = false;
            } else
```

```
        ++divisor;

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

    return esPrimo;
}

public Boolean call() throws Exception {

    return esPrimo(numero);
}

}

public class VentanaConCallable {

    private JFrame frame;
    private JTextField textField;

    /**
     * Launch the application.
     */
    public static void main(String[] args) {
        EventQueue.invokeLater(new Runnable() {
            public void run() {
                try {
                    VentanaConCallable window = new Ven
tanaConCallable();
                    window.frame.setVisible(true);
                } catch (Exception e) {
                    e.printStackTrace();
                }
            }
        });
    }
}
```

```
/**
 * Create the application.
 */
public VentanaConCallable() {
    initialize();
}

/**
 * Initialize the contents of the frame.
 */
private void initialize() {
    frame = new JFrame();
    frame.setBounds(100, 100, 838, 540);
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.getContentPane().setLayout(null);

    JLabel lblNewLabel = new JLabel("Resultado:");
    lblNewLabel.setFont(new Font("Tahoma", Font.PLAIN, 14));
    lblNewLabel.setBounds(32, 60, 150, 26);
    frame.getContentPane().add(lblNewLabel);
    textField
        = new JTextField();
    textField.setBounds(250, 39, 96, 19);
    frame.getContentPane().add(textField);
    textField.setColumns(10);

    JLabel lblNewLabel_1 = new JLabel("Introduzca un número:");
    lblNewLabel_1.setFont(new Font("Tahoma", Font.PLAIN, 14));
    lblNewLabel_1.setBounds(32, 37, 150, 19);
    frame.getContentPane().add(lblNewLabel_1);

    JLabel lblNewLabel_2 = new JLabel("");
    lblNewLabel_2.setBounds(250, 69, 217, 17);
```

```
frame.getContentPane().add(lblNewLabel_2);

JButton btnNewButton = new JButton("Calcular");
btnNewButton.setFont(new Font("Tahoma", Font.PLAIN, 14));
btnNewButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {

        lblNewLabel_2.setText("Esperando resultado ");
        int num = Integer.valueOf(textField.getText());

        Future<Boolean> tareaCallable1 = ForkJoinPool.commonP
ool().submit(new PrimoCallable2(num) );

        while (!tareaCallable1.isDone() && !tareaCallable1.isCancelled()) {

            System.out.println("Esperando Resultado");
        }

        try {
            if (tareaCallable1.get()) {
                lblNewLabel_2.setText("El numero es primo");

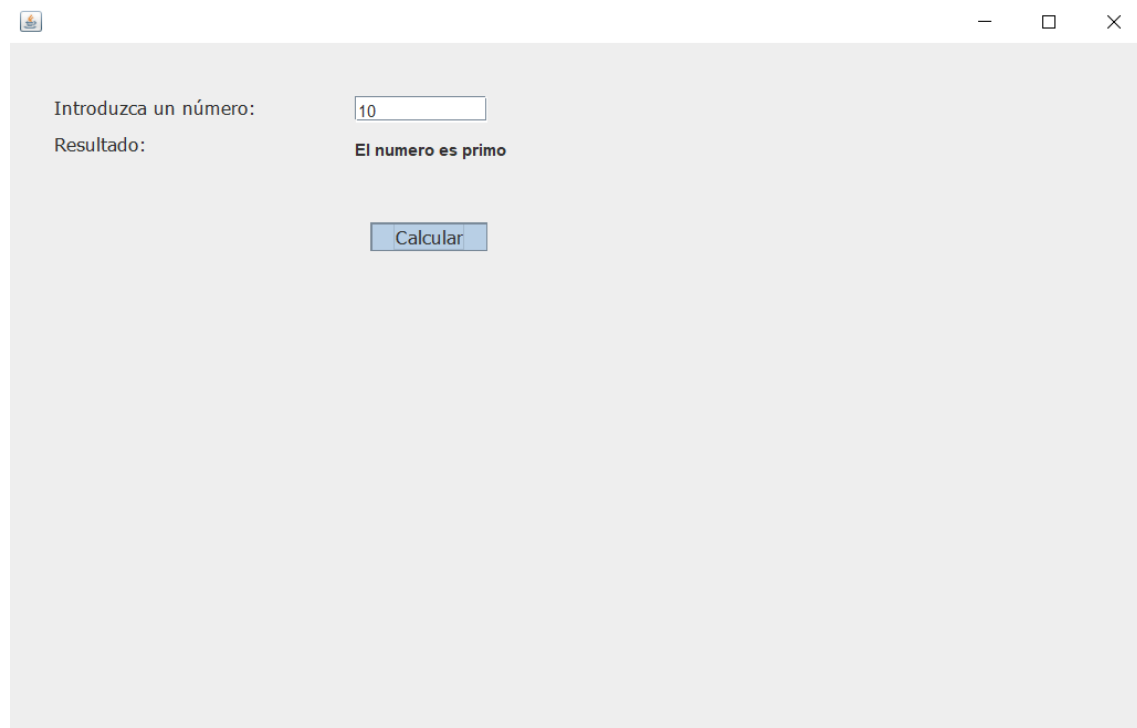
            } else {

                lblNewLabel_2.setText("El numero no es primo");
            }
        } catch (InterruptedException e1) {
            // TODO Auto-generated catch block
            e1.printStackTrace();
        } catch (ExecutionException e1) {
            // TODO Auto-generated catch block
            e1.printStackTrace();
        }

    }
}
```

```
        });  
  
        btnNewButton.setBounds(261, 130, 85, 21);  
        frame.getContentPane().add(btnNewButton);  
    }  
}
```

El **problema** que os encontrareis al ejecutar esta **versión síncrona de las futures** es **que bloquea el programa, el botón hasta que la future ha terminado**. Esto es debido a que por su **naturaleza síncrona, debemos esperar a que termine su ejecución** para poder recoger el resultado. Por eso vamos a **introducir las CompletableFuture** en este tema.



Comprobareis cuando realicemos un ejemplo similar con la CompletableFuture que esto no sucede. Puedo **lanzar la ejecución y además mientras espero mi programa no esta bloqueado.**

3.10 Usando el fork join pool con Callables

Esperamos el resultado de callables que van ejecutándose paralelamente, podemos usar el **nuevo pool con Callables**, en el caso de que quisiéramos compartirlo **con Hilos por ejemplo en un programa complejo**. Aquí os dejo un ejemplo demostrativo, de que el nuevo Pool también funciona con Callables.

```
import java.util.ArrayList;
import java.util.Random;
import java.util.concurrent.Callable;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ForkJoinPool;
import java.util.concurrent.ForkJoinTask;

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;
            }
            divisor++;
        } while (continuar);
        return esPrimo;
    }
}
```

```
        } else
            ++divisor;

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

    return esPrimo;

}

public Boolean call() throws Exception {

    return esPrimo(numero);

}

}

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

    public static void main(final String[] arguments) throws InterruptedException,
        ExecutionException {

        ArrayList<Integer> listaNumeros = new ArrayList<Integer>();
        ForkJoinPool forkJoinPool = new ForkJoinPool(4);
        int resultado=0;
        ForkJoinTask<Boolean> tareaCallable1;
        boolean esprimo = false;
        boolean bifurca= true;

        Random ran = new Random();

        for(int i=1; i<20 ; i++) {

            listaNumeros.add(ran.nextInt(30000));
```

```
    }

    for(Integer num : listaNumeros) {

        tareaCallable1 = forkJoinPool.submit(new PrimoCallable(num) );

        while (!tareaCallable1.isDone() && !tareaCallable1.isCancelled())

            {

                System.out.println("Esperando el resultado");

            }

            tareaCallable1.fork();

            if (tareaCallable1.join()) {

                System.out.println("El numero " + num + " es primo ");

            } else {

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

            }

        }

    }

}
```


3.11 Ejemplo con paralelismo hilos y Streams

En este ejemplo realizamos **paralelismo con hilos** como se realizaría en la actualidad. Es **válido sólo para los estudiantes que saben Streams**, para el resto es orientativo. **Sólo comentar que construimos un pool de 4**, y ejecutamos dos hilos que realizan el trabajo de manera paralela usando el **método `parallelStream()`**. `numbers.parallelStream()`

`HilosParalelosConStreams.java`

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;
import java.util.concurrent.ForkJoinPool;

public class HilosParalelosConStreams {

    private static List<Integer> buildIntRange() {
        List<Integer> numeros = new ArrayList<>(5);
        for (int i = 0; i < 100 ;i++)
            numeros.add(i);
        return Collections.unmodifiableList(numeros);
    }

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

        List<Integer> numeros = buildIntRange();

        ForkJoinPool forkJoinPool = new ForkJoinPool(4);

        Thread t1 = new Thread(() -> forkJoinPool.submit(() ->{

            numeros.parallelStream().forEach(n -> {
```

```
        try {

            Thread.sleep(50);

            System.out.println("Bucle 1 : " + Thread.currentThread());
;

        } catch (InterruptedException e) {

        }

    });

}).invoke();

ForkJoinPool forkJoinPool2 = new ForkJoinPool(4);

Thread t2 = new Thread(() -> forkJoinPool2.submit(() ->{

    numeros.parallelStream().forEach(n -> {

        try {

            Thread.sleep(50);

            System.out.println("Bucle 2 : " + Thread.currentThread())
;

        } catch (InterruptedException e) {

        }

    });

}).invoke());

t1.start();
```

```
        t2.start();

        t1.join();

        t2.join();

    }

}
```

3.12 Comparativa de velocidades de Streams paralelos y secuenciales. Obligatorio probarlo.

Por último, os dejo un **ejemplo de comparativa de paralelismo y secuencialidad en Streams**. Con este ejemplo quiero que **os hagáis una idea de las capacidades paralelas de Java 8, la API Stream, y el modelo fork join**. Probadlo.

ParalelismoStreamsComparativa.java

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;
import java.util.concurrent.TimeUnit;

public class ParalelismoStreamsComparativa {

    private static List<Integer> buildIntRange() {
        List<Integer> numbers = new ArrayList<>(5);
        for (int i = 0; i < 6000 ;i++)
```

```
        numbers.add(i);
    }
    return Collections.unmodifiableList(numbers);
}

public static void main(String[] args) {
    List<Integer> source = buildIntRange();

    long start = System.currentTimeMillis();
    for (int i = 0; i < source.size(); i++) {
        try {
            TimeUnit.MILLISECONDS.sleep(1);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
    System.out.println("Modo tradicional: " + (System.currentTimeMillis() - start) + "ms");

    start = System.currentTimeMillis();
    source.stream().forEach(r -> {
        try {
            TimeUnit.MILLISECONDS.sleep(1);
        } catch (Exception e) {
            e.printStackTrace();
        }
    });
    System.out.println("stream Con procesado secuencial: " + (System.currentTimeMillis() - start) + "ms");

    start = System.currentTimeMillis();
    source.parallelStream().forEach(r -> {
        try {
            TimeUnit.MILLISECONDS.sleep(1);
        } catch (Exception e) {
```

```
        e.printStackTrace();
    }
    });
    System.out.println("parallelStream :, Con procesado paralelo " +
        (System.currentTimeMillis() - start) + "ms");
}

}
```

4 Completable Futures

Esta **versión de Futures** son muy parecidas a las **Future Task** que vimos en el **tema anterior**, con la **diferencia** de que nos permiten **ejecuciones asíncronas y paralelas**, ya que **hace un uso transparente**, lo maneja **Java internamente**, del **jork join pool anterior**. Todos **nuestros servidores actuales son paralelos**, de esta manera **aprendemos los principios básicos del paralelismo java**. La **moderna API Stream** que os he proporcionado en los **apuntes de Java 8 y 9 Funcional**, basa **también su paralelismo** en este **pool**.

CompletableFuture se introduce en **Java 8**. Por defecto se **apoya en el framework fork-join** para lanzar su ejecución. Por tanto, estas **tareas en principio se ejecutan con paralelismo** y de **manera transparente al usuario**. También **podemos lanzarlas con un ExecutorService** pasado al final del método **sino deseamos paralelismo** y deseamos **usar el pool visto en el tema anterior**.

Encontrareis la información en la **siguiente página de Oracle**:

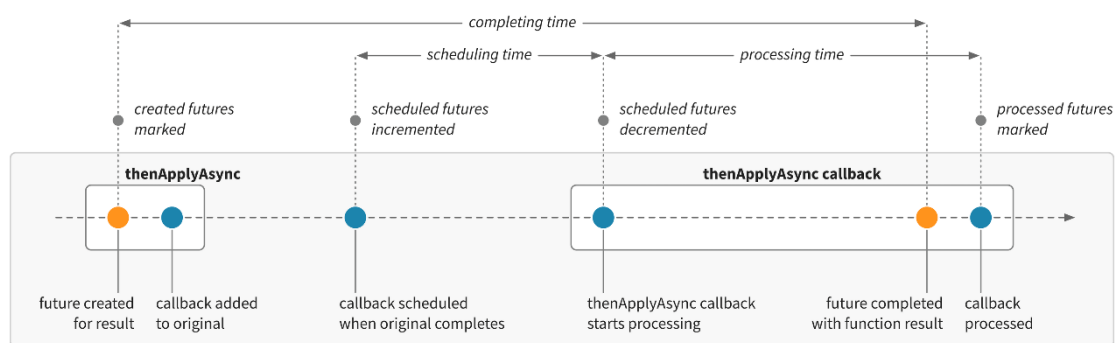
<https://docs.oracle.com/javase/10/docs/api/java/util/concurrent/CompletableFuture.html>

Las **CompletableFutures** hacen uso de **características de Java 8 y programación funcional** como los **interfaces funcionales y expresiones lambda**. Se va a proporcionar a **dar un uso básico para que entendáis dos conceptos nuevos, asincronía y callbacks**.

Las **CompletableFuture** pueden ser ejecutadas de manera asíncrona, esto quiere decir que nuestro hilo o programa que las ha llamado no va a esperar su resultado. A parte de ejecutarse en un hilo a parte, se comportan de manera independientes.

La pregunta es, ¿si nuestro programa no espera un resultado como lo obtenemos? Se obtiene por medio de uso de **Callbacks**. Una **Callback** es una función que se ejecuta cuando termina otra función. Entonces, vamos a configurar nuestra **CompletableFuture** con un **callback**. Cuando haya terminado de ejecutar la petición inicial se lanzará automáticamente otra función, que recogerá el resultado y realizará un trabajo con él.

En el siguiente gráfico podéis como funciona una **callback**. Después de llamar a **ApplyAsync** o **SupplyAsync** que será la ejecución principal de nuestra **completable future**, donde se ha creado para devolver un resultado, cuando se obtiene ese resultado de **ApplyAsync**, se llama al **callback thenApplyAsync**. Primero se añade la **callback** a la **future**, cuando se completa la **future** original se llama a la **callback**. Es como funcionan los eventos en la mayoría de los lenguajes de programación. Cuando se produce el evento tenemos preparado un **listener**, una **callback** que se va a ejecutar.



En este **primer ejemplo** vamos a probar como usar una **CompletableFuture** con una **ventana**, simulando una llamada que requiere mucho tiempo, como si nos respondiera un servidor, para ver su utilidad, que es el **funcionamiento asíncrono**. Con estas **futures** no bloqueamos nuestro programa como con las anteriores.

Este ejemplo es muy parecido al anterior que hicimos con la **Future** normal. Una **ventana**, pulsamos un botón que llama a una **CompletableFuture**, simula un **retardo de cinco segundos** y **calcula el factorial de un número**, mostrando el resultado por pantalla.

Vamos a centrarnos en el método **onclick** que es el que posee nuestra **lógica de negocio**.

En la **etiqueta** que tenemos oculta le indicamos al cliente que estamos **esperando la respuesta del resultado**.

Después creamos la **CompletableFuture**. Fijaos que **ahora no la añadimos ningún Callable**, la creamos directamente usando **expresiones lambda e interfaces funcionales**. El interfaz que estamos usando se llama **Supplier**. Haceros a la idea de que **ahora usamos funciones como parámetros en Java** como se hace en otros

lenguajes como Javascript. Este interfaz **representa una función que va a ser sobrescrita con una expresión lambda.** Con las **expresiones lambda** hacemos **sobrescritura de métodos.**

Estamos **sobrescribiendo y ejecutando al tiempo el método supplyAsync**, con un Supplier. **Un Supplier es como una función que no recibe parámetros y devuelve un resultado.** Muy parecido a como **sobrescribimos los listener en Eclipse y en Android.** Es el mismo principio. En **esta función que es la que se va a ejecutar para completar la Completable future**, se **calcula el factorial del número de la caja de texto y se devuelve su resultado.** En el momento que se **ejecuta el SupplyAsync** sobrescrito con la expresión lambda **la tarea comienza a ejecutarse.**

```
CompletableFuture<Long> completable = CompletableFuture.supplyAsync(() -> {  
    Long resultado = 1L;  
  
    try {  
        TimeUnit.SECONDS.sleep(5);  
  
        int numero = Integer.valueOf(textField.getText());  
  
        resultado = Factorial(numero);  
  
    } catch (InterruptedException e1) {  
        throw new IllegalStateException(e1);  
    }  
  
    return resultado;  
})
```

Después **vamos a añadir la callback**, cuando la **Future principal se ejecute**, **automáticamente se llamará a esta función**, que **recibirá como parámetro el resultado de la Future**. Como **veis esta callback se implementa con el método thenAcceptAsync**, que **recibe como parámetro otro interfaz funcional**, un **Consumer**, una **función expresada con Expresión lambda** que recibe un parámetro realiza una acción o proceso pero no devuelve resultado.

Lo que haremos es colocar **el resultado de la Future, el factorial**, recogido **en una etiqueta**. Así **funcionan todos los interfaces de las aplicaciones Cliente Servidor** desarrolladas en cualquier empresa.

```
completable.thenAcceptAsync((resultado)-> {  
  
    lblNewLabel_2.setText("Resultado recibido:" + resultado);  
  
    });  
  
}
```

Listener para el boton

```
btnNewButton.addActionListener(new ActionListener() {  
    public void actionPerformed(ActionEvent e) {  
  
        lblNewLabel_2.setText("Esperando resultado");  
        CompletableFuture<Long> completable = CompletableFuture.suppl  
yAsync(() -> {  
            Long resultado =1L;  
  
            try {  
                TimeUnit.SECONDS.sleep(5);
```



```
int numero = Integer.valueOf(textField.getText());

resultado = Factorial(numero);

    } catch (InterruptedException e1) {
        throw new IllegalStateException(e1);
    }
    return resultado;
});

completable.thenAcceptAsync((resultado)-> {

    lblNewLabel_2.setText("Resultado recibido:" + resultado);

});

}

});
```

VentanaCompletableFuture.java

```
import java.awt.EventQueue;
import javax.swing.JFrame;
import javax.swing.JLabel;
import java.awt.Font;
```

```
import javax.swing.JTextField;
import javax.swing.JButton;
import java.awt.event.ActionListener;
import java.util.concurrent.CompletableFuture;
import java.util.concurrent.TimeUnit;
import java.awt.event.ActionEvent;

public class VentanaCompletableFuture {

    private JFrame frame;
    private JTextField textField;

    /**
     * Launch the application.
     */
    public static void main(String[] args) {
        EventQueue.invokeLater(new Runnable() {
            public void run() {
                try {
                    VentanaCompletableFuture window = new
w VentanaCompletableFuture();
                    window.frame.setVisible(true);
                } catch (Exception e) {
                    e.printStackTrace();
                }
            }
        });
    }

    /**
     * Create the application.
     */
    public VentanaCompletableFuture() {
        initialize();
    }
}
```

```
}

public Long Factorial(int n) {

    Long resultado=1L;

    for (int i=1; i<=n ; i++) {

        resultado= resultado*i;

    }

    return resultado;

}

/**
 * Initialize the contents of the frame.
 */
private void initialize() {
    frame = new JFrame();
    frame.setBounds(100, 100, 838, 540);
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.getContentPane().setLayout(null);

    JLabel lblNewLabel = new JLabel("Resultado:");
    lblNewLabel.setFont(new Font("Tahoma", Font.PLAIN, 14));
    lblNewLabel.setBounds(32, 60, 150, 26);
    frame.getContentPane().add(lblNewLabel);

    textField
        = new JTextField();
    textField.setBounds(250, 39, 96, 19);
    frame.getContentPane().add(textField);
    textField.setColumns(10);

    JLabel lblNewLabel_1 = new JLabel("Introduzca un número:");
```

```
lblNewLabel_1.setFont(new Font("Tahoma", Font.PLAIN, 14));
lblNewLabel_1.setBounds(32, 37, 150, 19);
frame.getContentPane().add(lblNewLabel_1);

JLabel lblNewLabel_2 = new JLabel("");
lblNewLabel_2.setBounds(250, 69, 217, 17);
frame.getContentPane().add(lblNewLabel_2);

JButton btnNewButton = new JButton("Calcular");
btnNewButton.setFont(new Font("Tahoma", Font.PLAIN, 14));
btnNewButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {

        lblNewLabel_2.setText("Esperando resultado");
        CompletableFuture<Long> completable = CompletableFuture.suppl
yAsync(() -> {

            Long resultado = 1L;

            try {
                TimeUnit.SECONDS.sleep(5);

                int numero = Integer.valueOf(textField.getText());

                resultado = Factorial(numero);

            } catch (InterruptedException e1) {
                throw new IllegalStateException(e1);
            }
            return resultado;
        });

        completable.thenAcceptAsync((resultado)-> {
```

```
        lblNewLabel_2.setText("Resultado recibido:" + resultado);


    });

}

});

btnNewButton.setBounds(261, 130, 85, 21);
frame.getContentPane().add(btnNewButton);
}
}
```

Podeis comprobar en su ejecución que ni el botón ni el programa se bloquea, que sigue funcionando y podemos hacer otras cosas mientras en nuestro programa. Son las ventajas de la ejecución asíncrona que se lleva haciendo mucho tiempo en web. Y ahora, a partir de Java 8 se puede hacer en interfaces de escritorio Swing de java.



Introduzca un número:

Resultado: **Esperando resultado**

5 Modelos de diseño de servidores.

Al **principio del tema** veíamos **diseños arquitectónicos para servidores**. En esta parte final del tema **vamos a ver modelo de diseño de servidor para dar soluciones en la resolución de aplicaciones de tipo cliente servidor**. Puesto que en la asignatura de acceso a datos **veréis el modelo MVC**, y alguno más aquí vamos a ver un par de diseños diferentes. El primero de ellos va a ser el **de objeto activo**. De paso introduciremos una nueva clase en java que nos **permitirá mantener tareas en una cola, extraerlas y ejecutarlas una a una, con prioridades, la PriorityQueue**.

5.1 La BlockingQueue de java

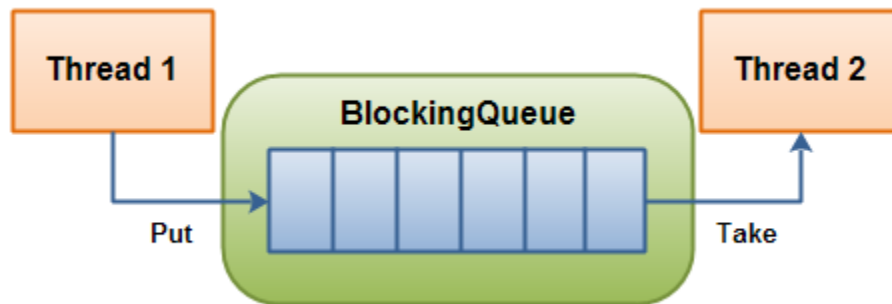
En el tema anterior **veíamos los tipos Atomic que eran lo que se conoce como thread-safe**. Esto quiere decir **que el acceso concurrente de los hilos se resuelve correctamente** como vimos en el tema anterior de concurrencia. Vamos a introducir otro **tipo de datos en java que nos proporciona una cola thread-safe**. Es decir, que **tiene implementada la exclusión mutua de tal manera que sólo un hilo puede acceder a la vez a ella**, y que **las operaciones de inserción y extracción de la cola se hacen en el orden en que se realizan las peticiones por los hilos**.

La **interfaz Java BlockingQueue**, `java.util.concurrent.BlockingQueue`, representa una **cola que es thread safe** para poner y sacar elementos de ella, esto quiere decir que tanto las **operaciones de insertar en cola como de sacar en cola son atómicas**. En otras palabras, varios hilos pueden insertar y tomar elementos simultáneamente desde Java `BlockingQueue`, **sin que surjan problemas de concurrencia**. **Sigue el modelo FIFO**, el **primero que entra es el primero que sale de la cola**.

El **término blocking proviene del hecho** de que Java `BlockingQueue` **es capaz de bloquear los hilos que intentan insertar o sacar elementos de la cola**. Por ejemplo, si **un hilo intenta sacar un elemento y no queda ninguno en la cola, el hilo se puede bloquear** hasta que haya un elemento que tomar. Que el hilo que realiza la llamada se **quede bloqueado o no depende de los métodos** que se llaman de `BlockingQueue`, como veremos a continuación.

Uso

Una **BlockingQueue** se utiliza normalmente **para hacer que un hilo produce objetos**, que otros hilos consumen. Lo podeis ver en el **siguiente diagrama**.



Vamos a **ver un sencillo ejemplo de productor -consumidor** como vimos en el **tema 2**, pero en vez de **implementar nosotros la concurrencia** usamos el **tipo BlockingQueue** de java.

El hilo productor **seguirá produciendo nuevos objetos e insertarlos en BlockingQueue**, hasta que **la cola alcance algún límite superior de elementos que puede contener**. Si la **cola de bloqueo alcanza su límite superior**, el **hilo productor se bloquea al intentar insertar el nuevo objeto**. Permanece **bloqueado** hasta que un **subproceso consumidor saca un objeto de la cola**.

El **hilo consumidor** sigue **quitando objetos de BlockingQueue** para procesarlos. Si el **hilo consumidor intenta sacar un objeto de una cola vacía**, el **consumidor se bloquea** hasta que un **hilo de producción coloca un objeto en la cola**.

La interfaz **Java BlockingQueue** tiene **4 conjuntos diferentes** de métodos para **insertar, eliminar y examinar los elementos** de la cola. Cada conjunto de métodos se comporta de manera diferente **en caso de que la operación solicitada no se pueda llevar a cabo inmediatamente**. Aquí está una tabla de los métodos:

	Lanza Exception	Valor Especial	Bloquea	Con Times Out	
Inserta	add(o)	offer(o)	put(o)	offer(o, timeout, timeunit)	
Quita	remove(o)	poll()	take()	poll(timeout, timeunit)	

Examina	element()	peek()			
---------	-----------	--------	--	--	--

Cuatro comportamientos diferentes para estas operaciones:

1. **Lanza una excepción:** Si la operación **intentada no es posible inmediatamente**, se **lanza una excepción**.
2. **Valor especial:** Si el **intento de operación no es posible** inmediatamente, se devuelve un **valor especial (normalmente true / false)**.
3. **Bloqueo:** Si la **operación intentada no es posible inmediatamente**, la llamada al **método se bloquea** hasta que lo es.
4. **Tiempos de espera:** Si la **operación intentada no es posible** inmediatamente, la llamada al **método se bloquea** hasta que lo es, pero **no espera más que el tiempo de espera** dado. Devuelve un valor especial que indica si la operación se realizó correctamente o no (normalmente true / false).

Vamos a ver el ejemplo brevemente es muy sencillo y a probarlo. **BlockingQueue** es un interfaz **necesitamos declarar una clase, vamos a usar ArrayBlockingQueue** una **implementación del interfaz BlockingQueue**

```
BlockingQueue cola = new ArrayBlockingQueue<Integer>(1024);
```

En productor y consumidor usamos put para meter elementos en la cola y take para sacar de cola. Son bloqueantes, mirad en la ejecución como **el consumidor se bloquea esperando a que el productor ponga** elementos en la cola.

```
System.out.println("Empezamos a poner");
cola.put(1);
```

```
System.out.println("Consumidor consume: " + cola.take());
```

EjemploProdConsBlockingQueue.java

```
import java.util.concurrent.ArrayBlockingQueue;
import java.util.concurrent.BlockingQueue;

public class EjemploProdConsBlockingQueue {

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

        BlockingQueue cola = new ArrayBlockingQueue<Integer>(1024);

        Productor productor = new Productor(cola);
        Consumidor consumidor = new Consumidor(cola);

        new Thread(productor).start();
        new Thread(consumidor).start();

        Thread.sleep(4000);
    }
}
```

productor.java

```
import java.util.concurrent.BlockingQueue;

public class Productor implements Runnable{

    protected BlockingQueue<Integer> cola = null;

    public Productor(BlockingQueue<Integer> cola) {
        this.cola = cola;
    }

    public void run() {
        try {
```

```
        System.out.println("Empezamos a poner");
        cola.put(1);
        System.out.println("Productor pone 1");
        Thread.sleep(1000);
        cola.put(2);
        System.out.println("Productor pone 2");
        Thread.sleep(1000);
        System.out.println("Productor pone 3");
        cola.put(3);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
}
```

consumidor.java

```
import java.util.concurrent.BlockingQueue;

public class Consumidor implements Runnable{

    protected BlockingQueue<Integer>    cola = null;

    public Consumidor(BlockingQueue<Integer>    cola) {
        this.colas = cola;
    }

    public void run() {
        try {
            System.out.println("Consumidor consume: " + cola.take());
            System.out.println("Consumidor consume: " + cola.take());
            System.out.println("Consumidor consume: " + cola.take());
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

```
    }  
  }  
}
```

5.2 La priority blocking queue

Es una version que implementa el interfaz anterior pero que además **da la posibilidad de incorporar prioridades**. La prioridad **puede ser incorporada de dos maneras**. La primera **usar el orden de prioridad de los elementos** que añadimos, de menor a mayor, menor el elemento, mayor la prioridad. Si insertamos un 1 es más prioritario que un 3.. La **segunda añadiendo un interfaz comparator, para comparar entre elementos**. Vamos a **ver esta cola sobre el ejemplo de Active Object** que incorporaremos a continuación.

Vamos a ver un sencillo ejemplo de la **PriorityBlockingQueue** y luego la **incorporaremos a nuestro programa Cliente Servidor** Active Object.

Declaramos la cola de tipo Integer en este caso y de tamaño 1024

```
PriorityBlockingQueue<Integer> cola = new PriorityBlockingQueue<Integer>(1024  
);
```

EjemploProdConsPriorityBlockingQueue.java

```
import java.util.concurrent.ArrayBlockingQueue;  
import java.util.concurrent.BlockingQueue;  
import java.util.concurrent.PriorityBlockingQueue;  
  
public class EjemploProdConsPriorityBlockingQueue {
```

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

            PriorityBlockingQueue<Integer> cola = new PriorityBlockingQueue<Integer>(1024);

            Productor productor = new Productor(cola);
            Consumidor consumidor = new Consumidor(cola);

            new Thread(productor).start();
            new Thread(consumidor).start();

            Thread.sleep(4000);
        }
    }
```

Productor.java

```
import java.util.concurrent.BlockingQueue;

public class Productor implements Runnable{

    protected BlockingQueue<Integer> cola = null;

    public Productor(BlockingQueue<Integer> cola) {
        this.cola = cola;
    }

    public void run() {
        try {

            System.out.println();
            cola.put("3");
            System.out.println("Productor pone 3");
            Thread.sleep(1000);
        }
    }
}
```

```
        cola.put("2");
        System.out.println("Productor pone 2");
        Thread.sleep(1000);
        System.out.println("Productor pone 1");
        cola.put("1");
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
}
```

Consumidor.java

```
import java.util.concurrent.BlockingQueue;

public class Consumidor implements Runnable{

    protected BlockingQueue<Integer> cola = null;

    public Consumidor(BlockingQueue<Integer> cola) {
        this.cola = cola;
    }

    public void run() {
        try {

            Thread.sleep(4000);
            System.out.println("Consumidor consume: " + cola.take());
            Thread.sleep(1000);

            System.out.println("Consumidor consume: " + cola.take());
            Thread.sleep(1000);
            System.out.println("Consumidor consume: " + cola.take());
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

```

    }
}
}

```

El ejemplo es prácticamente igual, pero **ahora el productor pone en cola 3 2 y 1**. Después, **el consumidor empieza a consumir**. Si os fijáis en la siguiente ejecución, **aunque 3 se ha introducido antes que 2 y 1, 3 es el ultimo elemento consumido**. Es por la **prioridad**, son **más prioritarios los elementos de menor tamaño**.

Ejecucion

```

Productor pone 3
Productor pone 2
Productor pone 1
Consumidor consume: 1
Consumidor consume: 2
Consumidor consume: 3

```

Podemos cambiar esto **pasandole un interface comparable a la cola** en su creación. Os lo **proporcione también en este otro ejemplo**.

```

import java.util.concurrent.BlockingQueue;
import java.util.concurrent.PriorityBlockingQueue;

public class EjemploProdConsPriorityBlockingQueueComparable {

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

        PriorityBlockingQueue<Integer> cola = new PriorityBlockingQueue<Integer>(1024, (e1,e2)-> e1>e2?-1:(e1<e2?1:0));

        Productor productor = new Productor(cola);
        Consumidor consumidor = new Consumidor(cola);

        new Thread(productor).start();
        new Thread(consumidor).start();

        Thread.sleep(4000);
    }
}

```

```
}
```

Comparable es un interfaz que devuelve 1 si el elemento 1 es mayor que el elemento 2. Devuelve -1 si el elemento 2 es mayor que el 1 y 0 si son iguales.

Hemos pasado un nuevo interfaz comparator con con **una expresión lambda** $(e1, e2) \rightarrow e1 > e2 ? -1 : (e1 < e2 ? 1 : 0)$; para que sea al contrario, como veis si el elemento e1 es mayor que el 2, devuelve -1. La finalidad es **que a mayor tamaño el elemento tenga mayor prioridad**.

Usando este último programa la ejecución nos devolvería:

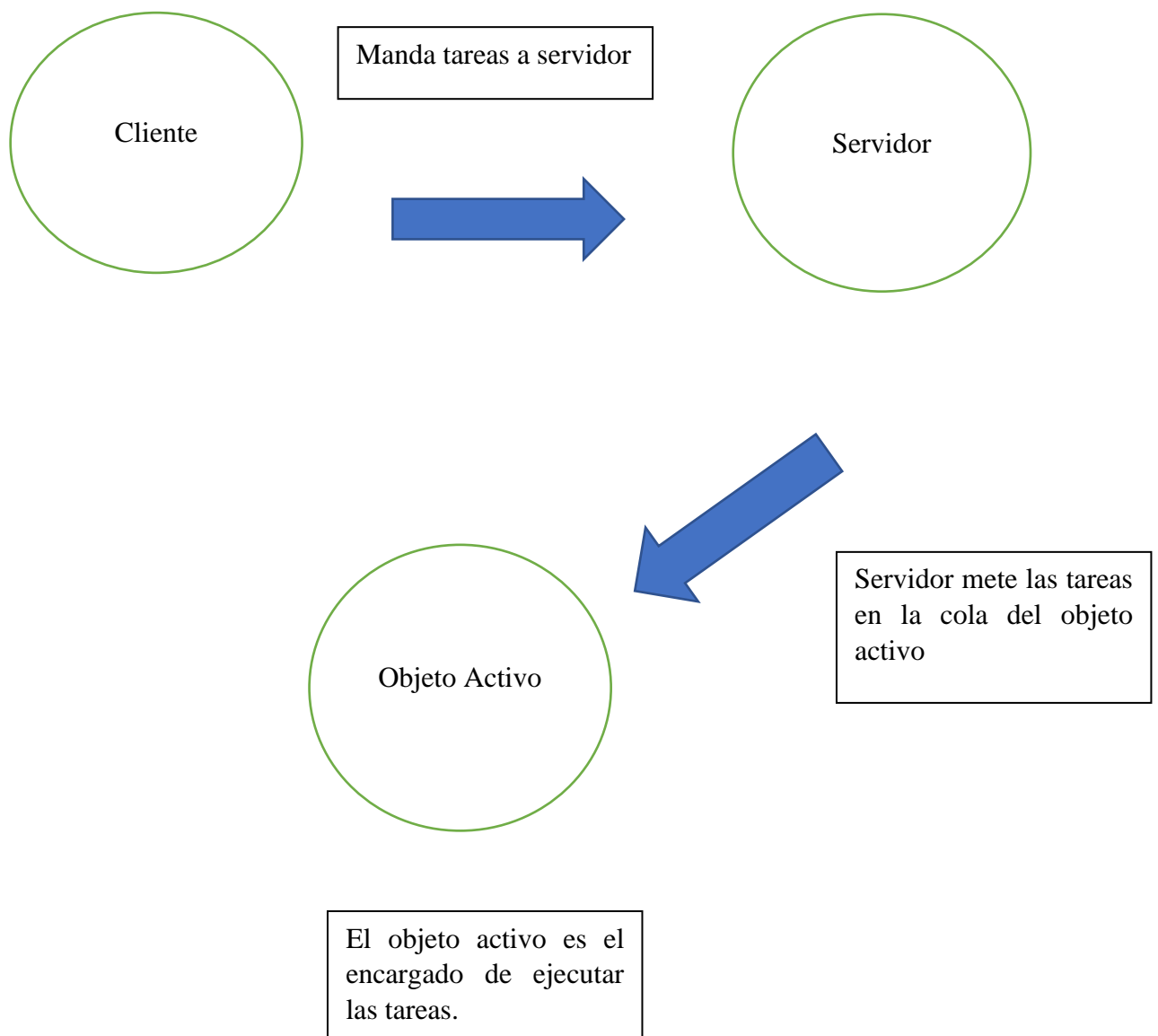
```
Productor pone 3
Productor pone 2
Productor pone 1
Consumidor consume: 3
Consumidor consume: 2
Consumidor consume: 1
```

5.3 Servidor Active Object

Este **tipo de servidores se cuenta con uno o mas clientes** que pueden **realizar peticiones de tareas** a uno o mas servidores. Estos **servidores redirigirán las tareas** a un Active Object o **Objeto Activo** que será el **encargado de ejecutar las tareas** por orden de llegada y/o prioridad. Para **guardar las tareas se puede usar una cola**. Nosotros **usaremos la PriorityQueue** de java. De esta manera nos **aseguramos un cierto orden** en la realización de **las tareas pedidas por nuestros clientes**.

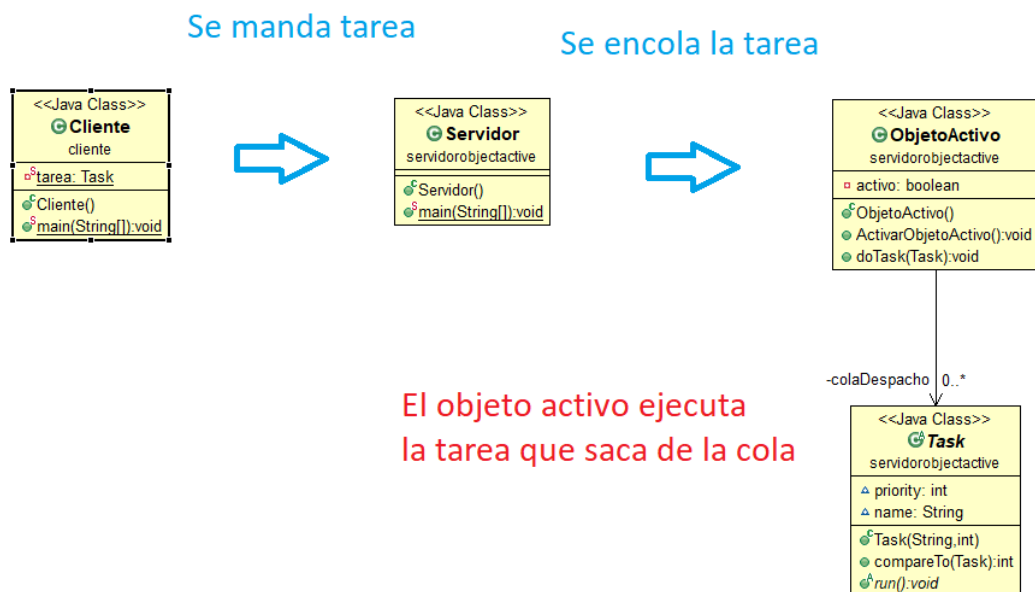
Este modelo es implementado **por muchos servidores y sistemas operativos** como **Windows , su servicio de nombres**, el servicio LDAP. Nosotros vamos a realizar una implementación donde **un cliente manda tareas a un servidor** con un socket en un objeto de tipo Task, y **el servidor las redirige al Objeto Activo, guardándolas en la cola del Objeto Activo**. El Objeto Activo saca una a una de la cola las tareas, teniendo en cuenta su prioridad y orden de llegada, y las ejecuta. Lo que **guardaremos en la cola será una clase Task**, que **implementa runnable**, es un hilo, y además **tiene definido un atributo prioridad**. Task es un hilo para poder ser ejecutada.

El modelo de servidor se ilustra en el siguiente gráfico:



Es parecido a un **dispatcher de tareas**, un **repartidor de tareas**, pero nos da la ventaja de **poder añadir varios clientes y varios servidores que redirigan las tareas al objeto activo**. Que la cola de prioridades **sea thread safe**, hace que cada operación **de inserción por parte de los hilos sea segura y atómica**.

El modelo de clases sería el siguiente



La tarea, clase Task

La clase Task es una clase que implementa runnable, un hilo, y tiene su atributo **prioridad**. Para ese atributo **hemos implementado el interfaz comparable**, de manera que **a mayor prioridad, el elemento es considerado como de menor tamaño**. Ya sabéis que la PriorityQueue ordena **la entrada de elementos de menor a mayor**. Para que la **prioridad mayor sea mas prioritaria** ha de ser **considerada como más pequeña**.

```

public int compareTo(Task other) {
    return (this.priority>other.priority?-1:(this.priority<other.priority)?1:0);
}
  
```

Implementamos runnable para que se pueda ejecutar como un hilo y **Serializable** para que puede ser pasado **a través de un socket al servidor. Hemos hecho que la clase sea abstracta**, esto es porque vamos a definir el método run sobrescribiendolo en el cliente con una clase anónima.

```
public abstract void run() ;
```

```
import java.io.Serializable;

public abstract class Task implements Runnable, Serializable, Comparable < Task > {

    int priority;
    String name;

    public Task(String name, int priority) {
        this.name = name;
        this.priority = priority;
    }

    public int compareTo(Task other) {
        return (this.priority>other.priority?-1:(this.priority<other.priority)?1:0);
    }

    public abstract void run() ;

}
```

El cliente:

Usa un socket para enviar la tarea Task al servidor.

```
Socket socketEscribeServer = new  
Socket(Constants.Host, Constants.PuertoEscrituraCliente);
```

Crea las tareas en tiempo de ejecución con una clase anónima sobrescribiendo el método run(). La primera tarea muestra los 100 primeros números del 0 al 99 y tiene prioridad 3.

```
Task tarea1= new Task("muestra numeros", 3) {  
  
    public void run() {  
        for (int i=0; i<100; i++) {  
            System.out.println(" Numero: " +i);  
        }  
    }  
};
```

La segunda tarea suma los 100 primeros números y tiene prioridad 4.

```
Task tarea2= new Task("muestra suma numeros", 4) {  
  
    public void run() {  
  
        int suma=0;  
  
        for (int i=0; i<100; i++) {  
  
            suma+=i;  
  
        }  
  
        System.out.println(" Suma  Nùmeros: " +suma)  
;  
  
    }  
  
};
```

Estas **dos tareas no las ejecuta el cliente, se las manda al servidor por medio del socket y el servidor se encarga de insertarlas en la cola del objeto activo.**

```
ObjectOutputStream output = new ObjectOutputStream(socketEscribeServer.getOutputStream());  
  
    output.writeObject(tarea1);  
  
    output.close();  
  
    socketEscribeServer.close();
```

```
        socketEscribeServer = new Socket(Constants.Host, Constantes.PuertoEscrituraCliente);

        output = new ObjectOutputStream(socketEscribeServer.getOutputStream());

        output.writeObject(tarea2);
```

Cliente.java

```
import java.io.ObjectOutputStream;
import java.net.Socket;

import servidorobjectactive.Task;

public class Cliente {

    public static void main(String args[]) {

        try {
            Socket socketEscribeServer = new Socket(Constants.Host, Constantes.PuertoEscrituraCliente);

            Task tarea1= new Task("muestra numeros", 3) {

                public void run() {

                    for (int i=0; i<100; i++) {

                        System.out.println(" Numero: " +i);

                    }

                }

            }

        }

    }

}
```

```
        }

    };

Task tarea2= new Task("muestra suma numeros", 4) {

    public void run() {

        int suma=0;

        for (int i=0; i<100; i++) {

            suma+=i;

        }

        System.out.println(" Suma  Nùmeros: " +suma)
;

    }

};

ObjectOutputStream output = new ObjectOutputStream(socketEscrib
ibeServer.getOutputStream());

output.writeObject(tarea1);

output.close();
```

```
        socketEscribeServer.close();

        socketEscribeServer = new Socket(Constants.Host, Constantes.PuertoEscrituraCliente);

        output = new ObjectOutputStream(socketEscribeServer.getOutputStream());

        output.writeObject(tarea2);

        output.close();

        socketEscribeServer.close();

    } catch (Exception e) {

        e.printStackTrace();

    }

}

}
```

Usamos un **fichero de constantes** para definir los puertos y la IP del servidor:

Constantes.java

```
public class Constantes {

    public void Constantes() {
```



```
    }

    public static final int PuertoEscrituraCliente=6000;
    public static final int PuertoLecturaCliente=6001;

    public static final String Host= "localhost";

}
```

El servidor

El servidor se **va a encargar de** crear el objeto activo **recibir peticiones por el socket**, **lanzar un hilo que resuelva la petición**, y este **hilo recogerá las tareas enviadas por los clientes y las meterá en la cola de peticiones** del objeto activo. Para ello usamos el **nuevo pool ForkJoinPool**, para paralelizar.

```
        ForkJoinTask<?> tareapool=null;
        HiloPeticiones petition= null;
        Socket cliente=null;
        ServerSocket servidor= new ServerSocket(Cons
tantes.PuertoEscrituraCliente);
        ForkJoinPool pool= new ForkJoinPool(4);
```

Como veis el servidor tiene un bucle infinito, **recibe peticiones por el socket**, y **lanza hilos para resolverlas**. En el momento que **se recibe la primera petición se activa el Objeto activo**, esta **acción es realizada para probar y que se vea el funcionamiento las prioridades en el ejemplo**, no haría falta.

```
while (true) {
```

```
cliente= servidor.accept();  
obj.ActivarObjetoActivo();
```

Cuando acepta la petición crea un hilo para resolver la petición, le pasa el objeto activo, el socket cliente, la conexión del cliente y lo ejecuta con el submit. No necesitamos hacer get, porque no necesitamos un resultado de vuelta, es un hilo.

```
peticion=new HiloPeticiones(cliente, obj);  
tareapool= pool.submit(peticion);
```

Servidor.java

```
import java.io.IOException;  
import java.net.ServerSocket;  
import java.net.Socket;  
import java.util.concurrent.ForkJoinPool;  
import java.util.concurrent.ForkJoinTask;  
  
public class Servidor {  
  
    public static void main(String args[]) {  
  
        ObjetoActivo obj = new ObjetoActivo();
```

```
// Call doTask in different threads

    try {

        ForkJoinTask<?> tareapool=null;
        HiloPeticiones peticion= null;
        Socket cliente=null;
        ServerSocket servidor= new ServerSocket(Cons
tantes.PuertoEscrituraCliente);
        ForkJoinPool pool= new ForkJoinPool(4);

        while (true) {

            cliente= servidor.accept();
            obj.ActivarObjetoActivo();
            peticion=new HiloPeticiones(cliente
, obj);
            tareapool= pool.submit(peticion);

        }

    } catch (Exception e) {

        // TODO Auto-generated catch block
        e.printStackTrace();

    }

}

}
```

Hilo que resuelve peticiones

La **tarea del hilo petición** será la de **resolver las peticiones del cliente** y liberar de trabajo al servidor, que ya puede estar atendiendo otras peticiones del cliente. Lo que hace **este hilo es coger la tarea enviada por el cliente en un socket y ponerla en la cola del objeto activo**. El servidor nos ha pasado el socket de conexión con el cliente, lo usamos para obtener la tarea enviada por el cliente y ponerla en la cola del objeto activo

```
streamLectura = new ObjectInputStream(cliente.getInputStream());
```

```
tarea = (Task) streamLectura.readObject();
```

```
object.doTask(tarea);
```

```
import java.io.ObjectInputStream;
import java.net.Socket;
import java.util.concurrent.ForkJoinPool;

public class HiloPeticiones extends Thread{
    private Socket cliente;
    private Task tarea;
    private ObjetoActivo object;
```

```
public HiloPeticiones(Socket cliente, ObjetoActivo objact) {

    this.cliente=cliente;

    this.objact=objact;

}

public void run() {

    Task tarea=null;
    ObjectInputStream streamLectura;
    try {
        streamLectura = new ObjectInputStream(cliente.getInputStream());
        // 
        // 
        tarea = (Task) streamLectura.readObject();
        // 
        objact.doTask(tarea);

    } catch (Exception e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    }

}

}
```

El objeto activo

El objeto **activo** se encarga de ofrecer los servicios para insertar en la cola de prioridades las tareas **Task** y ejecutarlas.

Declaramos una **cola de prioridades de tamaño 20**

```
private PriorityBlockingQueue < Task > colaDespacho = new PriorityBlockingQueue < Task > (20);
```

Ofrecemos un método **para activar el objeto activo**. Cuando se activa lanza un hilo que entra **en un bucle que básicamente coge tareas** de la cola **con el método take** y las ejecuta.

```
if (!activo) {
    activo=true;
    new Thread(() -> {
        while (activo) {
            try {

                Thread.sleep(2000);

                System.out.println("Tamaño de la cola" + colaDespacho.size(
));
                Task task = colaDespacho.take();

                System.out.println("Ejecuta tarea de nombre " + task.name
+ " y prioridad " + task.priority);
                new Thread(task).start();

            } catch (InterruptedException e) {
                break;
            }
        }
    })
```

```

        }
    })
    .start();

}

}

```

Ofrece otro método para **meter tareas a ejecutar en la cola**, el **doTask**. **HiloPetición** usará este método para poner tareas enviadas por el cliente en la cola.

```

public void doTask(Task tarea) {
    colaDespacho.put(tarea);
}

```

```

import java.util.concurrent.PriorityBlockingQueue;

public class ObjetoActivo {

    private boolean activo=false;

    private PriorityBlockingQueue < Task > colaDespacho = new PriorityBlockingQ
ueue < Task > (20);

    public ObjetoActivo() {

    }

    public void ActivarObjetoActivo() {

        if (!activo) {
            activo=true;
            new Thread(() -> {

```

```
        while (activo) {
            try {

                Thread.sleep(2000);

                System.out.println("Tamaño de la cola" + colaDespacho.size(
));

                Task task = colaDespacho.take();

                System.out.println("Ejecuta tarea de nombre " + task.name
+ " y prioridad " + task.priority);
                new Thread(task).start();

            } catch (InterruptedException e) {
                break;
            }
        }
    })
    .start();

}

}

public void doTask(Task tarea) {
    colaDespacho.put(tarea);
}
}
```


Dos detalles de la cola de prioridades

1. Si la cola se queda vacía no hay problema, el hilo del objeto activo se quedará bloqueado a la espera de nuevas tareas. Cuando llegue una tarea nueva se desbloquea.
2. Cuando la cola está llena, las nuevas tareas que lleguen se descartan, hay que tener muy claro que tamaño tiene que tener nuestra cola de antemano, en caso contrario podemos perder tareas por el camino.

5.4 Ejercicio

Modificar el servidor para:

1. Que por medio de la consola ofrezca la posibilidad de pararlo cuando introduzcamos una F, mayúscula. Deberá parar primero los hilos que ha lanzado, el objeto activo, y finalmente a sí mismo.
 - a. Parar hilos de ejecución HiloPetición
 - b. Desactivar Objeto Activo
 - c. Parar Servidor.

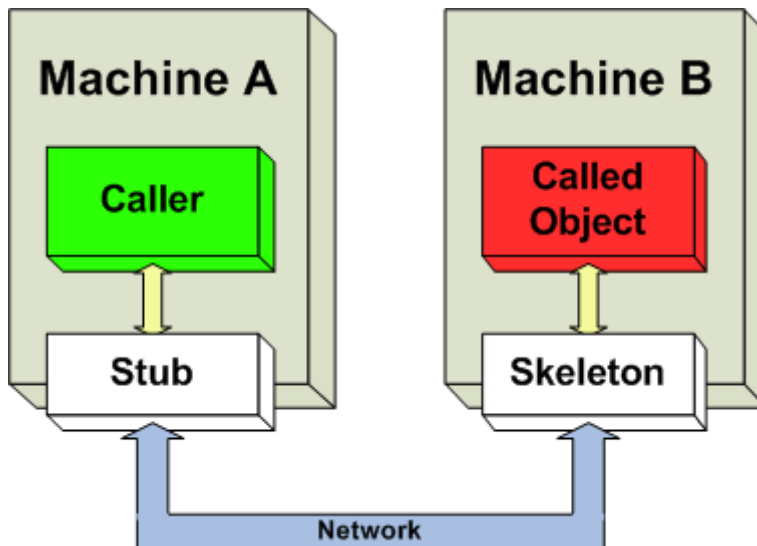
Añadir un nuevo cliente:

2. Introducir otro cliente que mande dos tareas. Una suma de números impares del 1 al 200 prioridad 5, y el cálculo del factorial para el 40, prioridad 2.

Ejecutar todo a la vez para probarlo

5.5 Modelo de servidor objeto Stub

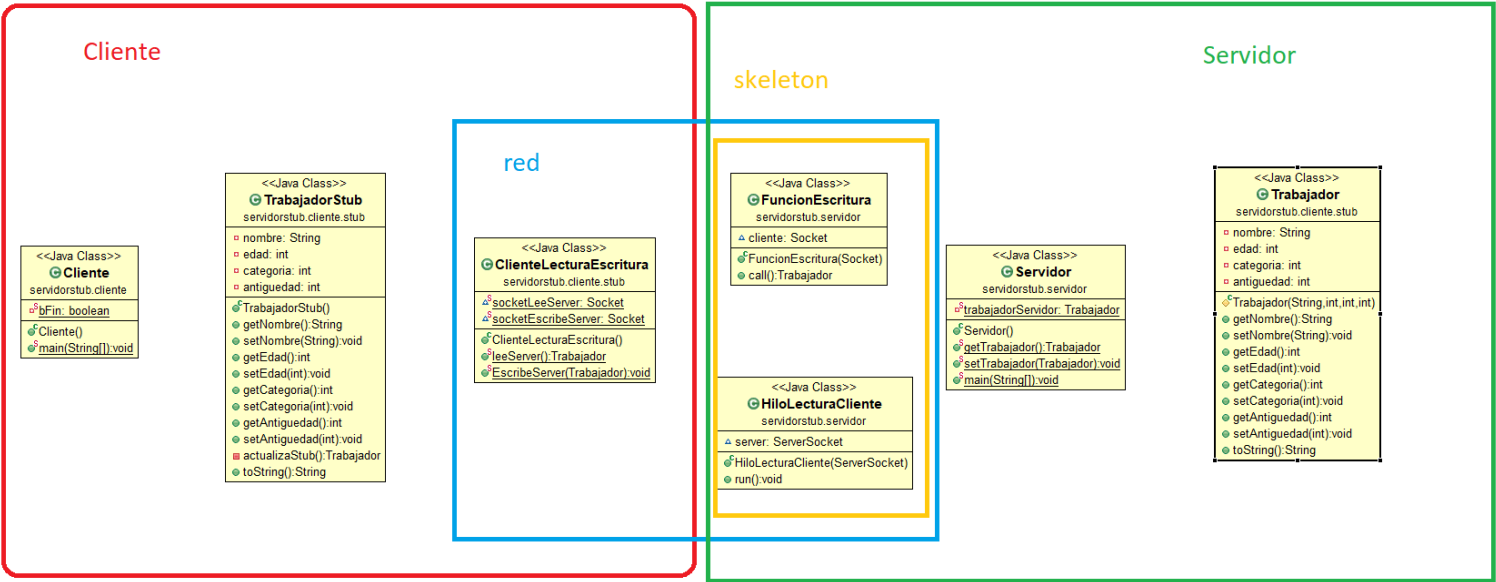
Para finalizar el **tema vamos a realizar un último modelo de servidor**. Se conoce como **Modelo Stub**. La idea es la de poder **manejar objetos de servidor como si estuvieran en cliente de manera transparente**. Para ello se proporciona un mecanismo u objeto Stub. Mediante este objeto **Stub vamos a manejar el objeto servidor como si lo tuviéramos en cliente**, pero **usando la red y el protocolo TCP IP para manejar ese objeto servidor**. Realizamos una versión sencilla de Stub en la que usando los getters vamos a leer de las propiedades del objeto servidor y con los setter vamos a escribir en el objeto que esta en el servidor. De esta manera creamos la falsa sensación de que el cliente **maneja un objeto de servidor como si fuera un objeto local**.



Como podéis ver en la **figura se ofrece un Stub**, que nos dará la sensación de manejar el **objeto servidor como local**, un **Skeleton o esqueleto en servidor** que nos **permite procesar esas llamadas**. La **red entre medias para realizar todas las comunicaciones necesarias para llevar a cabo esta función**.

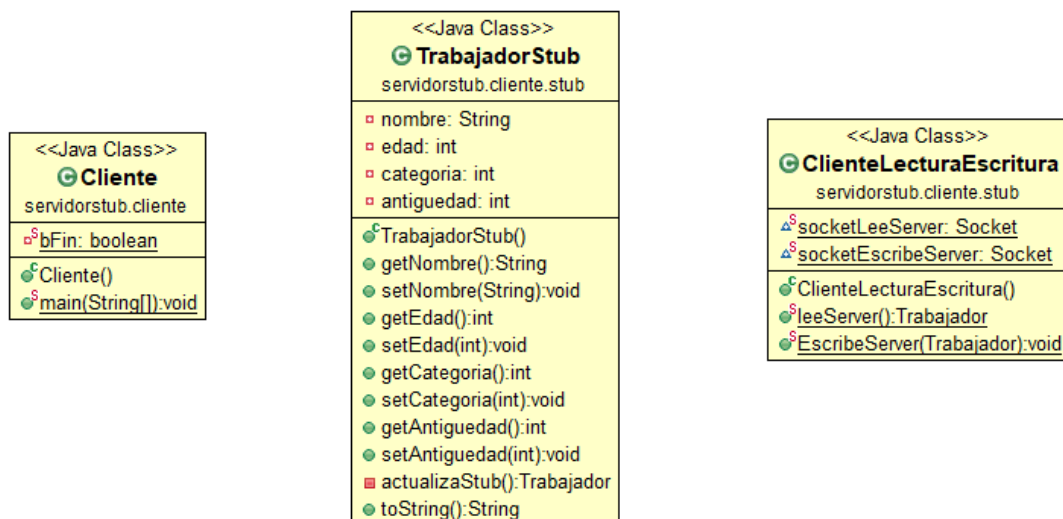
5.6 Nuestro modelo

Este **modelo** y su código se explicará en detalle en clase. El código se proporcionará en el proyecto **ServidorStub** . Básicamente **hemos establecido las clases en cliente y en servidor** que se encargarán de realizar el **Stub**, la **parte de red** y de esqueleto y nuestro modelo de datos.



La parte cliente

En la parte cliente tenemos **un cliente que intenta manejar el objeto de servidor Trabajador a través del TrabajadorStub**. Una clase **ClienteLecturaEscritura** que maneja las comunicaciones con el servidor.



La parte de servidor

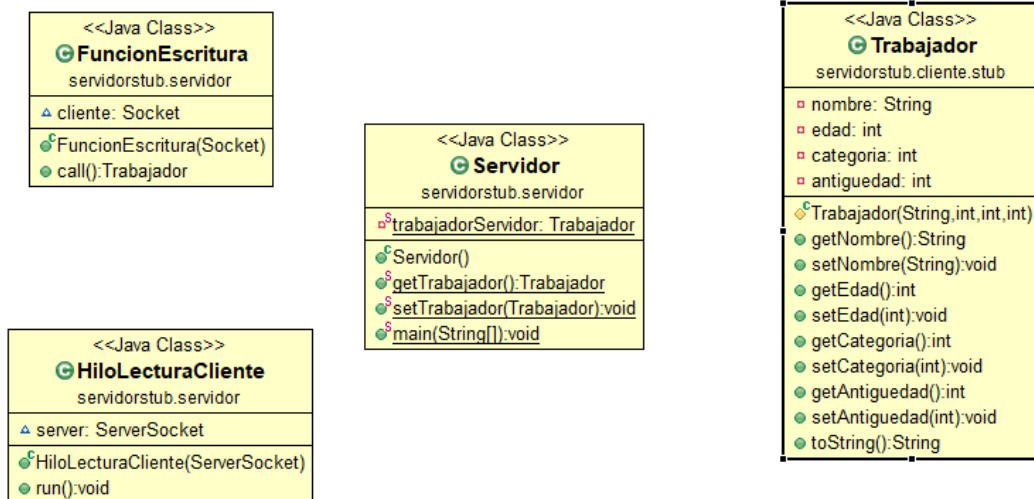
FunciónEscritura, una **Future**, e **HiloLecturaFuncion**, un **hilo**, que se encargan de las comunicaciones en red, de ofrecer lecturas y escrituras en el **Trabajador** con **sockets**. Podrían ser considerados el esqueleto de nuestro modelo.

El **Servidor** recoge las peticiones de cliente y las atiende, usando las dos clases anteriores, de manera paralela, con el pool **Fork Join**. Y **trabajador** es nuestro modelo de datos, el que guarda el estado del trabajador, ofrece **getters** y **setters**,

Servidor ofrece dos métodos, uno `getTrabajador`, para que el cliente lea el trabajador entero, y otro `setTrabajador`, para que cambie el trabajador. Hemos establecido que cada vez que llamemos a un getter desde `TrabajadorStub`, leeremos el objeto entero `Trabajador`, así nos aseguramos tener la última versión de servidor de trabajador. Si hacemos `getNombre()` nos traemos el trabajador entero, y devolvemos el atributo nombre.

Cuando hacemos un setter, leemos el trabajador entero, para tener la última versión y cambiamos sólo el campo que nos interese. Por ejemplo, si hacemos `setNombre` desde `TrabajadorStub`, leemos el `Trabajador` entero de servidor y luego cambiamos el nombre y lo escribimos en servidor.

No es la implementación perfecta, pero es un ejercicio práctico de clase para que entendáis este nuevo modelo.



Propuestas de mejora:

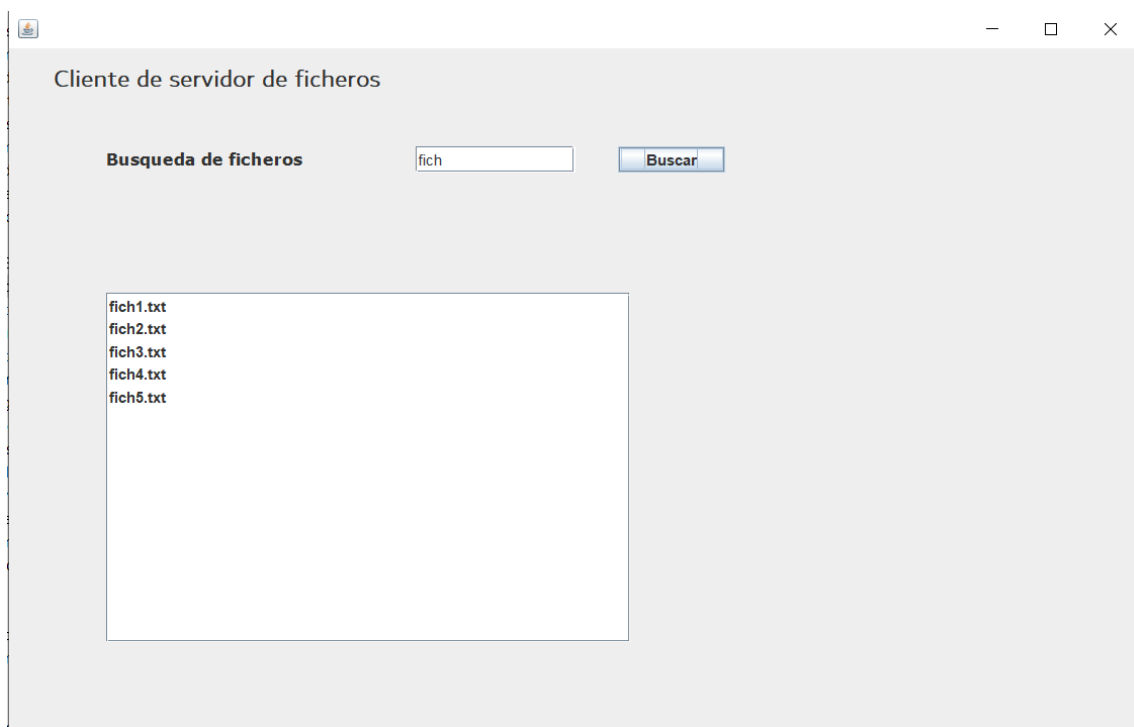
1. Podríamos **introducir un objeto activo que hiciera las transacciones en orden, con una cola de prioridades.**
2. Introducir **una prioridad en manera de marca de tiempo.** Obtenemos el **tiempo del sistema en milisegundos.** Cuanto **más antigua la transacción se realiza antes.**
3. Introducir **más clientes y/o más servidores.**

6 Servidor de ficheros. Tarea

En este tema se proporcionará un servidor de ficheros. **Esta implementada la primera parte, la búsqueda de ficheros.** Deberéis implementar **la segunda, la descarga de ficheros.** Estamos **realizando una versión sencilla del Emule, sin Peer to Peer** ya que ofrecemos **además de servicio de búsqueda en servidor como el Emule, el servicio de descarga también será en servidor.**

Se **proporciona el código de búsqueda con dos puertos UDP.** Deberéis **realizar la descarga de ficheros en cliente con un puerto TCP.** La **descripción de los pasos a realizar se proporciona a continuación.** Primero ilustraremos el programa inicial cliente.

Este es el cliente.



1. **Estudiareis el código** por vuestra cuenta.
2. **Mejorar el interfaz a vuestro gusto.**
3. Deberemos **modificarlo para sustituir la Future por una CompletableFuture.**
4. Añadir los **elementos necesarios para realizar una descarga:**
 - a. La **descarga se llevará a una ventana nueva** que también tendréis que crear.