

WUOLAH



raulcb98

www.wuolah.com/student/raulcb98



2620

Apuntes Prácticas PCD - Estructuras de sincronización.pdf

Estructuras de sincronización en Java



3º Programación Concurrente y Distribuida



Grado en Ingeniería Informática



**Escuela Técnica Superior de Ingeniería
UHU - Universidad de Huelva**

Como aún estás en la portada, es momento de redes sociales. Cotilléanos y luego a estudiar.



Wuolah

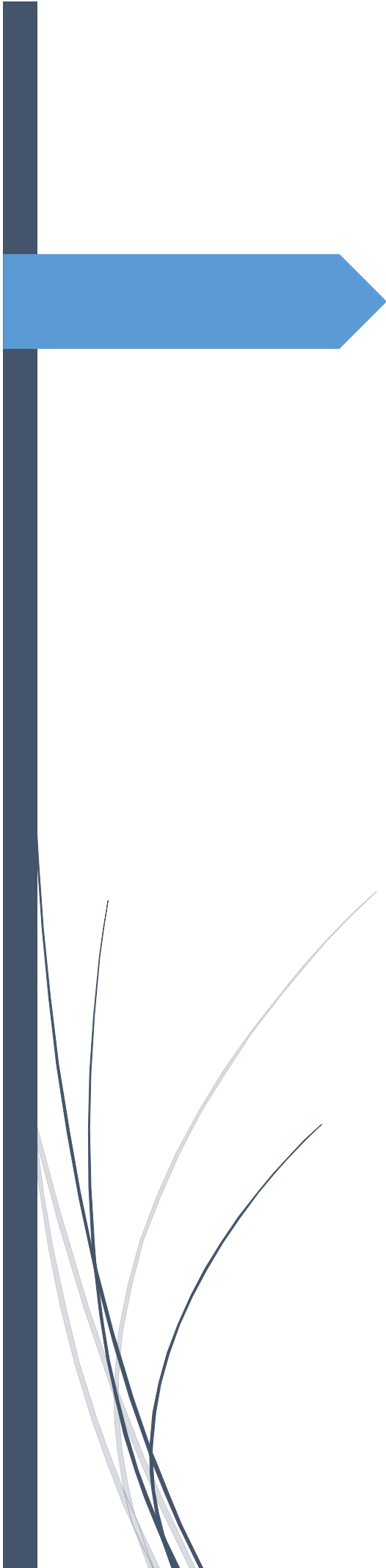


Wuolah



Wuolah_apuntes

WUOLAH



Programación Concurrente y Distribuida

Apuntes Prácticas

Raúl Castilla Bravo
3º INGENIERÍA INFORMÁTICA
ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA (ETSI – UHU)



SIN ACEITE DE PALMA



**LA MEJOR RECOMPENSA DESPUÉS
DE UNA TARDE DE ESTUDIO**

¿NOCILLEAMOS?

nociyeah!

ÍNDICE

HILOS EN JAVA.....	pag. 2
I. <u>Heredando de la clase Thread</u>	pag. 2
II. <u>Implementando la interfaz Runnable</u>	pag. 2
CREAR UNA APPLET	pag. 3
I. <u>Clase Applet</u>	pag. 3
ESTRUCTURA BÁSICA DE UN PROGRAMA CON CANVAS	pag. 4
I. <u>Clase Canvas</u>	pag. 4
II. <u>Clase Applet</u>	pag. 5
MÉTODOS SINCRONIZADOS.....	pag. 6
I. <u>Sincronizar bloques de código</u>	pag. 6
II. <u>Errores con synchronized</u>	pag. 9
SEMÁFOROS EN JAVA	pag. 10
I. <u>Semáforo general</u>	pag. 10
II. <u>Semáforo binario</u>	pag. 10
III. <u>Clase Semaphore</u>	pag. 11
MONITORES: REENTRANTLOCK	pag. 12
I. <u>Monitores para problema Lectores - Escritores</u>	pag. 12
II. <u>Monitores con variables Condition</u>	pag. 15
THREADPOOL.....	pag. 17
I. <u>Con Runnable</u>	pag. 17
II. <u>Con Callable</u>	pag. 18
RMI (REMOTE METHOD INVOCATION).....	pag. 20
I. <u>Interfaz del recurso</u>	pag. 21
II. <u>Implementación de la interfaz del recurso</u>	pag. 21
III. <u>Implementación del servidor</u>	pag. 22
II. <u>Implementación del cliente</u>	pag. 23

HILOS EN JAVA

Los hilos en Java están asociados a una clase, es decir, nuestros hilos van a ser “objetos” de una clase con la peculiaridad de que solo ejecutan un método llamado `run()`. Lo que escribamos en el método `run()` serán aquellas acciones que queremos que se ejecuten de forma concurrente.

Tenemos dos formas de implementar las clases en Java para que permitan hilos:

- Que la clase herede de la clase `Thread`.
- Que la clase implemente la interfaz `Runnable`.

Sea cual sea el método que usemos, los hilos van a actuar de la misma forma.

Heredando de la clase `Thread`

Si queremos que la clase herede de `Thread`, la declaración de la clase sería la siguiente:

```
public class HiloThread extends Thread{  
  
    @Override  
    public void run() {  
  
    }  
  
}
```

La declaración de un hilo sería igual que si declaramos un objeto.

```
HiloThread h1 = new HiloThread();
```

Implementando la interfaz `Runnable`

Si queremos que la clase implemente la interfaz `Runnable`, la declaración de la clase sería la siguiente:

```
public class HiloRunnable implements Runnable {  
  
    @Override  
    public void run() {  
  
    }  
  
}
```

La declaración de un hilo consiste en declarar un objeto de la clase `Thread`:

```
Thread h4 = new Thread(new HiloRunnable());
```

Las operaciones que se pueden ejecutar sobre un hilo son: `start()` para que comience su ejecución y `join()` para esperar a que finalice un hilo (no para hacer que un hilo espere).

Programación Concurrente y Distribuida (PCD)

CREAR UNA APPLET

Applet es una librería que permite establecer una conexión cliente servidor. Nosotros utilizaremos de esta librería las herramientas gráficas que ofrece. Un Applet es un contenedor que solo muestra una ventana. Lo que se visualiza dentro de esa ventana es un Canvas, un gráfico sobre el que podemos pintar.

Por tanto, para hacer una Applet en Java, necesitamos dos clases, una para el Applet (la ventana) y otra para el Canvas (el gráfico). Por ejemplo, las clases podrían ser:

- MiCanvas: es una clase que hereda de Canvas.
- mainApplet: es una clase que hereda de mainApplet.

La clase MiCanvas se crea como cualquier otra clase en Java: New -> Java Class. Sin embargo, mainApplet se crea con: New -> Applet. Si no encontramos la opción Applet, buscamos en Others. Para este tipo de aplicaciones también nos valdría una JApplet.

Clase Applet

La clase Applet será la clase principal de nuestra aplicación. Los métodos que contiene son:

```
public class mainApplet extends Applet {  
  
    public void init() {  
  
    }  
  
    @Override  
    public void start() {  
  
    }  
  
    @Override  
    public void stop() {  
  
    }  
  
}
```

La clase Applet no tiene un main(), sino un init(). El init() se ejecuta al momento de comenzar al igual que pasaría con un main(). En el init() se establecen los parámetros de la ventana (tamaño, color...) y se crean las variables que vayamos a usar durante la ejecución.

El método start() será aquello que se ejecute cuando finalice el init() y el método stop() se ejecuta cuando se cierra la aplicación.

ESTRUCTURA BÁSICA DE UN PROGRAMA CON CANVAS

Un programa con un Canvas tiene dos clases al menos:

- Clase canvas que hereda de Canvas.
- Clase applet que hereda de Applet.

Para que se realice correctamente la visualización, debemos tener implementada la siguiente estructura:

Clase Canvas

```
public class CanvasLE extends Canvas{

    //Constructor de ventana Canvas
    public CanvasLE(int ancho, int alto){
        setBackground(Color.white);
        setSize(ancho, alto); //Dimensiones
    }

    //Método para actualizar los atributos del canvas (opcional)
    public void actualiza()

    }

    //Método para que se llama para repintar
    @Override
    public void update(Graphics g) {
        this.paint(g);
    }

    //Método para pintar
    @Override
    public void paint(Graphics g){
        //Obtenemos el gráfico para pintar
        Image offscreen = createImage(getWidth(), getHeight());
        Graphics og = offscreen.getGraphics();

        //Pintamos
        og.setColor(Color.red);
        og.fillOval(100, 100, 100, 100);

        //Mostramos en el Canvas
        g.drawImage(offscreen, 0, 0, null);
    }
}
```

Clase Applet

```
public class MainApplet extends Applet {

    @Override
    public void init() {
        CanvasLE canvas;
        int ancho, alto;

        //Inicializamos la ventana del Canvas
        ancho = 1000;
        alto = 500;

        setBackground(Color.white);
        setSize(ancho, alto);

        canvas = new CanvasLE(ancho, alto);
        add(canvas);

        //Creamos los hilos
    }

    @Override
    public void start() {
        //Lanzamos los hilos con start()
    }

    @Override
    public void stop() {
        //Interrumpimos los hilos con interrupt()
    }
}
```

Nota: Lo primero que se ejecuta es el `init()` y después el `start()`. No se visualiza nada en el canvas hasta que no se ha finalizado el método `start()`. Si tenemos algo que bloquee la ejecución antes de finalizar `start()`, no veremos nada visualizado en el canvas.

Nota(II): el método `stop()` se ejecuta cuando cerramos la ventana del canvas. Para finalizar forzosamente a los hilos, hacemos `interrupt()`. También podríamos usar un `join()` para esperar a que finalicen su ejecución de forma normal, pero la ventana del canvas no se cerrará hasta que no finalicen. El `interrupt()` es para que se cierre todo de inmediato.

MÉTODOS SINCRONIZADOS

Podemos garantizar exclusión mutua en un método utilizando la palabra clave `synchronized` de la siguiente forma:

```
public synchronized void Acola(Object elemento) {
    if(!colallena())
    {
        this.datos[this.tail] = elemento;
        this.tail++;
        this.tail = this.tail%this.capacidad;
        this.numelementos++;
    }
}
```

En este método solo puede haber un hilo ejecutándose y hasta que no termine con la última sentencia no podrá entrar otro.

Sincronizar bloques de código

Vamos a suponer un método que tiene mucho código pero no quiere sincronizar el código completo sino una parte. Eso quiere decir que no podemos sincronizar el método porque entonces el método va a estar sincronizado al completo.

Lo que tenemos que hacer es que sobre el bloque de código que queremos sincronizar le ponemos el `synchronized`.

```
synchronized() {

    //bloque de código sincronizado.

}
```

Esto implica que el método entero no está sincronizado, solo el trozo de código entre las llaves. Lo cual quiere decir que el resto de código del método puede ser concurrente. `synchronized` tiene como argumento el objeto con el que está sincronizándose.

```
synchronized(this) {

    //bloque de código sincronizado.

}
```

Formación
Online
Especializada

Clases Online
Prácticas
Becas

Ponle
nombre
a lo que
quieres ser

Jose María Girela
Bim Manager.

Programación Concurrente y Distribuida (PCD)

Podemos tener un método completo sincronizado y estaría sincronizado también con los bloques synchronizes.

```
class Recurso{
    public synchronized void metodo1 () {
        //Bloque de código
    }

    public void metodo2 () {
        synchronized(this){
            //Bloque de código
        }
    }

    public synchronized void metodo3 () {
        synchronized(this){
            //Bloque de código
        }
    }
}
```

Ahora mismo, solo podemos tener un hilo dentro de los trozos de metodo2, metodo3 o método1. Vamos a suponer que queremos que pueda haber un hilo diferente dentro de metodo1, metodo2 y metodo3. Para conseguir esto, lo que tenemos que hacer es no sincronizar los códigos sobre el mismo objeto.

```
class Recurso{
    public synchronized void metodo1 () {
        //Bloque de código
    }

    public void metodo2 (Recurso A) {
        synchronized(A) {
            //Bloque de código
        }
    }

    public synchronized void metodo3 (Recurso B) {
        synchronized(B) {
            //Bloque de código
        }
    }
}

main() {
    Object A = new Object();
    Object B = new Object();
}
```

En este caso, puede haber un hilo en cada método pero no dos hilos sobre el mismo método porque cada método está sincronizado de forma diferente.

- El método 1: está sincronizado con this.
- El método 2: está sincronizado con A.
- El método 3: está sincronizado con B.

En un trozo de código sincronizado tengo tres operaciones que puedo realizar:

- wait() -> El hilo que corre por ahí se queda parado (no tiene nada que ver con el wait() del semáforo).
- notify() -> Despierta a algún hilo que esté parado en el wait() pero notify() no se para.
- notifyAll() -> Despierta a todos los hilos que estén dormidos.

```
class Recurso{  
    public synchronized void metodo1 () {  
        wait ()  
    }  
  
    public synchronized void metodo2 () {  
        notify ()  
    }  
}
```

Cuando un hilo está parado con wait() en un trozo de código, se para fuera de la sincronización, es decir, permite que otro hilo entre y ejecute el código del método.

Cuando un hilo hace notify, no quiere decir que dos hilos vayan a correr a la vez, Java se encarga de que eso no pase.

Cualquier objeto que hereda de Object puede hacer wait() y notify(). Estos son métodos de un objeto. Si en un trozo de código no se especifica qué objeto ejecuta el wait(), es decir este caso:

```
public synchronized void metodo1 () {  
    wait ()  
}
```

Sería equivalente:

```
public synchronized void metodo1 () {  
    this.wait ()  
}
```

Errores con synchronized

Error 1: No se puede utilizar wait(), notify() o notifyAll() en un bloque que no esté sincronizado.

```
public void metodo1 () {  
    wait (); //Incorrecto  
}
```

Error 2: Solo se puede utilizar wait(), notify() o notifyAll() en un bloque de código que esté sincronizado sobre el objeto que ejecuta wait(), notify() o notifyAll().

```
class Recurso{  
    public void metodo2 () {  
        synchronized(A) {  
            A.wait (); //Correcto  
            B.notify (); //Incorrecto  
            wait (); //Incorrecto porque es this.wait()  
        }  
    }  
  
    public synchronized void metodo3 () {  
        synchronized(B) {  
            B.wait (); //Correcto  
            A.wait (); //Incorrecto  
        }  
    }  
}
```

Si hacemos wait() en un objeto distinto al que estoy sincronizado se bloquea la ejecución del método.

SEMÁFOROS EN JAVA

El código de un semáforo en Java es muy sencillo, así que a la hora de trabajar con semáforos podemos optar por utilizar la clase Semaphore de java o implementar nuestro propio semáforo. La implementación es la siguiente:

Semáforo general

```
public class SemaforoGeneral {  
  
    protected volatile int contador;  
  
    public SemaforoGeneral(int inicial) throws Exception {  
        if (inicial < 0) {  
            throw new Exception("Error");  
        }  
        contador = inicial;  
    }  
    public synchronized void WAIT() throws InterruptedException {  
        while (contador == 0) {  
            wait();  
        }  
        contador--;  
    }  
    public synchronized void SIGNAL() {  
        contador++;  
        notify();  
    }  
}
```

Semáforo binario

```
public class SemaforoBinario {  
    protected volatile int contador;  
  
    public SemaforoBinario(int inicial) throws Exception{  
        if (inicial != 0 && inicial != 1){  
            throw new Exception("Imposible inicializar");  
        }  
        contador = inicial;  
    }  
    public synchronized void WAIT() throws InterruptedException{  
        while(contador == 0){  
            wait();  
        }  
        contador = 0;  
    }  
    public synchronized void SIGNAL() {  
        contador = 1;  
        notify();  
    }  
}
```


Programación Concurrente y Distribuida (PCD)

Clase Semaphore Java

Las operaciones que podemos ejecutar con un semáforo de Java son:

```
Semaphore semaforo = new Semaphore(4); //Contador inicializado a 4  
  
semaforo.acquire(); //Equivalente a wait()  
semaforo.release(); //Equivalente a signal()
```

Formación
Online
Especializada

Clases Online
Prácticas
Becas

Ponle
nombre
a lo que
quieres ser

Jose María Girela
Bim Manager.



MONITORES: REENTRANT LOCK

En Java existe una interface llamada interface Lock que usaremos para definir un monitor en java. Lock es un interface, así que no podemos definir objetos de tipo Lock(). Java ya dispone de una implementación específica para el problema de lectores escritores llamada ReentrantLock. Esta clase deriva de Lock y de ella sí se pueden crear objetos.

Monitores para problema Lectores - Escritores

El ReadWriteLock es una interface (subtipo) que hereda de Lock que es capaz de determinar si la exclusión mútua se hace en lectura o en escritura. Necesitamos una implementación de esta interfaz que se llama ReentrantReadWriteLock().

```
ReadWriteLock rwl = new ReentrantReadWriteLock();
```

Con un objeto de este tipo podemos hacer exclusión mútua en una parte concreta del código. Para ello, hacemos:

```
rwl.lock();  
  
//(SECCIÓN EXCLUIDA)  
  
rwl.unlock();
```

Esta sección tiene exclusión mútua con cualquier otro trozo de código que tenga el lock en el objeto rwl. Lo que más nos interesa es que el bloque (SECCIÓN EXCLUIDA) se puede definir para lectura o para escritura de la siguiente forma:

```
rwl.readlock().lock(); //Protocolo entrada lectores  
  
//(SECCIÓN EXCLUIDA)  
  
rwl.readlock().unlock(); //Protocolo salida lectores
```

En la sección excluida se cumplen las condiciones de lectura del problema de los lectores. Es decir, en esa sección puede haber varios lectores pero un solo escritor. Cuando queremos acceder en escritura:

```
rwl.writelock().lock(); //Protocolo entrada escritores  
  
//(SECCIÓN EXCLUIDA)  
  
rwl.writelock().unlock(); //Protocolo salida escritores
```

A la hora de aplicar los protocolos de entrada, podemos seguir una política justa o injusta:

- Justa: entra el que lleva más tiempo esperando.
- Injusta: la entrada es aleatoria.

Estas políticas se aplican en el constructor al declarar el objeto y se asemejan a las prioridades de lectura-escritura (**nota:** se asemejan, no son iguales).

```
ReadWriteLock rwl = new ReentrantReadWriteLock(true); //Política justa
```

Si dentro de la sección excluida salta una excepción, nunca se ejecutaría el unlock(). Para evitar que cualquier ruptura o excepción deje bloqueado el método, encerramos toda la sección excluida en un bloque try catch.

```
rwl.readlock().lock(); //Protocolo entrada lectores

try{
    //(SECCIÓN EXCLUIDA)

} catch(...){
    ...
} finally{
    rwl.readlock().unlock(); //Protocolo salida lectores
}
```

Nota: El finally se ejecuta siempre, salte excepción o no.

Para utilizar correctamente este tipo de bloqueos, debemos de tener en cuenta las operaciones que estamos realizando dentro del bloque excluido. Por ejemplo:

```
rwl.readlock().lock(); //Protocolo entrada lectores

(SECCIÓN EXCLUIDA) //Por ejemplo

if(tengo dinero)
    sacardinero(); //Este método escribe en una variable

rwl.readlock().unlock(); //Protocolo salida lectores
```

Estamos intentando utilizar un método que escribe en una variable teniendo solo un bloqueo de lectura. Esto es incorrecto, primero tendríamos que obtener el bloqueo del recurso en escritura.

Para conseguir el bloque en escritura, hacemos lo siguiente:

```
rw1.readlock().lock(); //Soy lector

if(tengo dinero){
    rw1.readlock().unlock(); //Dejo de ser lector
    rw1.writelock().lock(); //Soy escritor

    sacardinero();

    rw1.readlock().lock(); //Soy lector
    rw1.writelock().unlock(); //Dejo de ser escritor
}

//Comprobar saldo

rw1.readlock().unlock(); //Dejo de ser lector
```

Si nos fijamos en el trozo de código, veremos que antes de dejar de ser escritores nos hemos hecho lectores aplicando el bloqueo de lectura. Esto se llama degradación de permiso (downgrade). Primero bloqueamos la lectura y luego soltamos la escritura.

Siempre se permite hacer un bloqueo de una petición de permisos menos restrictiva. De esta forma se evita que en el momento que dejo de ser escritor, otro hilo pase a ser escritor y yo me quede esperando intentando leer.

Nota: la degradación de permisos se hace desde escritura a lectura. Si intentamos lo contrario, reservar la escritura sin soltar la lectura nos bloquearíamos porque habría siempre un lector en el recurso y no se nos concedería el permiso de escritura.

Nota(II): Lo que diferencia esta práctica con la práctica 5 de los lectores y escritores es que en esta no vamos a tener un clase recurso con métodos para indicar el protocolo de entrada y salida de los lectores y escritores. En nuestro caso, vamos a tener un objeto `ReentrantReadWriteLock()` que ya contiene los protocolos de entrada y salida de los lectores y escritores.

Programación Concurrente y Distribuida (PCD)

Monitores con variables Condition

Hemos visto que con los ReentrantLock podíamos conseguir exclusión mutua de la siguiente manera:

```
void m1 ()
{
    bloqueo.lock ()

    ...Bloque excluido

    bloqueo.unlock ()
}

void m2 ()
{
    bloqueo.lock ()

    ...Bloque excluido

    finally
    bloqueo.unlock ()
}
```

Esto nos garantiza la exclusión mutua que obtendríamos como con un synchronize pero con la diferencia de que ahora podemos definir variables de tipo Condition. Las variables Condition se asignan a un ReentrantLock y cuando un hilo se duerme en un bloqueo, se acola en la variable Condition, de manera que podemos controlar qué hilos despertamos con facilidad.

Las condition no se pueden crear de forma aislada, siempre van asociadas a un ReentrantLock.

```
ReentrantLock bloqueo = new ReentrantLock ();
Condition c1 = bloqueo.newCondition ();
Condition c2 = bloqueo.newCondition ();
```


Para que funcione bien, debemos asegurarnos de que usamos la variable condición dentro del bloque excluido de la variable ReentrantLock. Las operaciones que podemos realizar con variables Condition son `await()` y `signal()`.

```
void m1()
{
    bloqueo.lock()
    try{
        c1.await(); //Me duermo en la cola de c1
        c2.signal(); //Despertamos un hilo de la cola de c2
    } finally {
        bloqueo.unlock();
    }
}

void m2()
{
    bloqueo.lock()
    try{
        c2.await(); //Me duermo en la cola de c2
        c1.signal(); //Despertamos un hilo de la cola de c1
    } finally {
        bloqueo.unlock();
    }
}
```

Nota: Si a la hora de declarar el ReentrantLock (boolean fair) lo ponemos a true, se intenta que el próximo en entrar sea el que lleve más tiempo.

THREADPOOL

Si creamos muchos hilos (muchísimos), el sistema no es capaz de soportar la creación de tantos hilos y se colapsa la aplicación. Para evitar esto, se limita el número de hilos que se pueden crear utilizando un ThreadPool.

Un ThreadPool es una bolsa de hilos que ya existe en el sistema y que se crea con una capacidad concreta que puede soportar el sistema. Si hay más peticiones que hilos y se usan todos los hilos, las peticiones se esperan hasta que haya hilos disponibles.

Un ThreadPool en Java es un `ExecutorService` y la forma de crearlo es la siguiente:

```
ExecutorService thp = Executors.newFixedThreadPool(2);
//Esto es un ThreadPool de 2 hilos
```

Con Runnable

Para explicar el funcionamiento del ThreadPool vamos a suponer el siguiente ejemplo:

```
class sinretorno implements Runnable {
    private int id;

    public sinretorno(int id) {
        this.id = id;
    }

    @Override
    public void run() {
        for (int i = 0; i < 10; i++) {
            System.out.println("Soy " + id);
        }
    }
}

main() {
    ExecutorService thp = Executors.newFixedThreadPool(2);

    for (int i = 0; i < 5; i++) {
        sinretorno r = new sinretorno(i);
        thp.execute(r);
    }
    thp.shutdown(); //Finaliza cuando finaliza la lista de tarea
}
```

La clase `sinretorno` implementa `Runnable` para que puedan crearse hilos que se puedan usar en el ThreadPool y en el `main` hemos creado un ThreadPool con dos hilos. Lo que se hace a continuación es crear cinco hilos e introducirlos en el ThreadPool con el método `execute()`. El método `execute()` no solo introduce los hilos sino que también los inicia.

Como el ThreadPool solo tiene dos hilos, habrá tres que se tengan que esperar a que finalicen los dos que han entrado primero y se les atenderá conforme vayan acabando las tareas.

Con Callable

Según las necesidades de la aplicación, puede ser necesario recoger algún valor cuando los hilos finalicen sus tareas en el ThreadPool. Para conseguir que el hilo devuelve un valor, es necesario que haya implementado Callable como vemos a continuación:

```
class conretorno implements Callable<Integer> {

    private int id;

    public conretorno(int id) {
        this.id = id;
    }
    @Override
    public Integer call() {
        for (int i = 0; i < 10; i++) {
            System.out.println("Soy " + id);
        }
        return id + 1000;
    }
}
```

Debemos prestar atención a algunos detalles:

- La clase implementa Callable y es de tipo Integer. Esto quiere decir que el call devuelve un Integer.
- El método que se debe sobrescribir no es run() sino call().
- call() devuelve un Integer.

El dato que devuelve cada hilo se debe almacenar en una variable de tipo Future. Future es una referencia hacia una zona de memoria.

```
ArrayList<Future<Integer>> f = new ArrayList<>();
for (int i = 0; i < 5; i++) {
    conretorno cr = new conretorno(i);
    f.add(thp.submit(cr));
}

for (int i = 0; i < f.size(); i++) {
    System.out.println("La tarea " + i + " devuelve " + f.get(i).get());
}

thp.shutdown(); //Finaliza cuando finaliza la lista de tarea
```

En el ejemplo hemos creado un array de objetos Future. Nótese que para añadir un hilo al ThreadPool ahora no usamos execute() sino submit(). El método submit() devuelve un objeto tipo Future que es el que se guarda en el array.

Una vez introducidos todos los hilos en el ThreadPool, se puede continuar la ejecución hayan o no finalizado los hilos. Solo se para la ejecución cuando intentamos utilizar el valor de un objeto Future y el hilo aún no ha puesto el valor en esa zona de memoria. El método que produce la espera es en nuestro caso f.get(i).get(). El primer get(i) es para obtener la variable Future del array y el segundo para obtener el valor almacenado en Future.

Programación Concurrente y Distribuida (PCD)

Siempre que terminemos de usar un ThreadPool debemos cerrarlo. Tenemos dos opciones:

```
thp.shutdown(); //Finaliza cuando finaliza la lista de tarea  
thp.shutdownNow(); //Finaliza ahora independientemente de lo que  
haya en la lista de tareas.
```

La primera cierra el ThreadPool cuando todas las tareas han sido atendidas y la segunda cierra el ThreadPool de forma abrupta, de manera que habrá tarea que queden sin atender.

Nota: execute() solo sirve para hilos de tipo Runnable, pero submit() sirve tanto para Runnable como para Callable.

Formación
Online
Especializada

Clases Online
Prácticas
Becas

Ponle
nombre
a lo que
quieres ser

Jose María Girela
Bim Manager.



RMI (Remote Method Invocation)

RMI se usa para comunicar aplicaciones Java que corren en máquinas distintas. Vamos a tener un servidor que va a crear un objeto y ese objeto lo va a hacer público a los clientes que se pueden comunicar con él a través de la red.

Solo existirá **un objeto**, que es mantenido por el servidor, y todos los clientes usarán ese objeto. De tal forma que cuando los clientes tengan acceso a la referencia de ese objeto, los clientes no crearán un objeto nuevo con new sino que usarán el objeto del servidor.

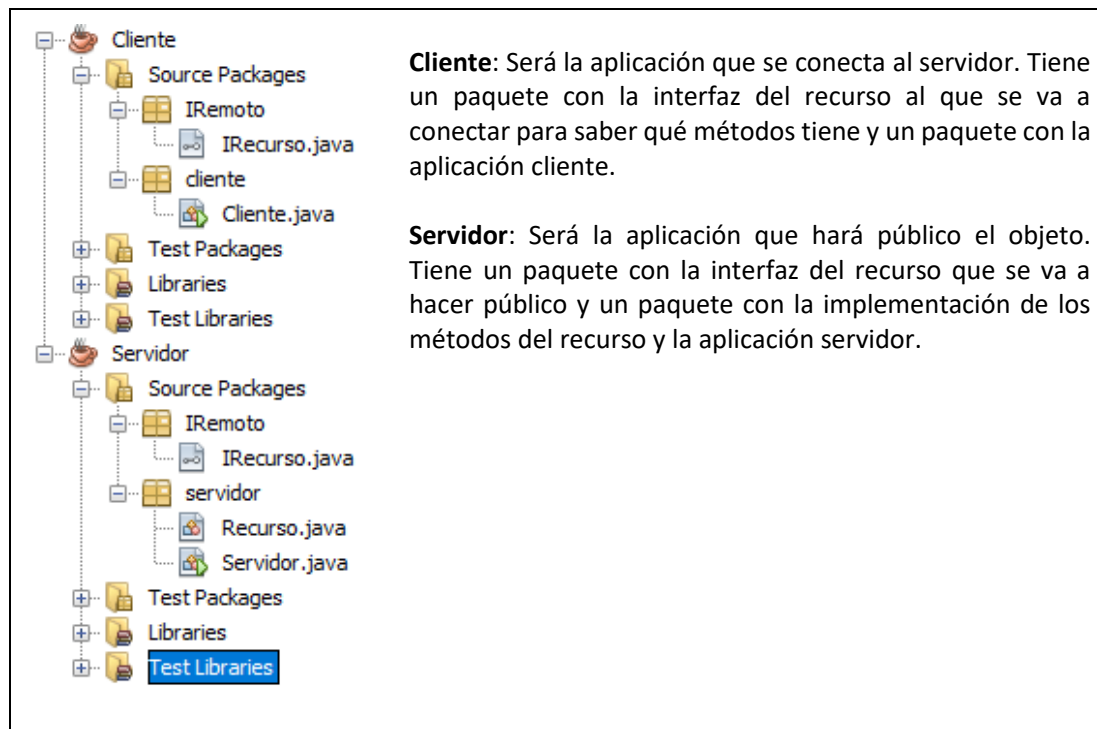
Para poder conectarse al objeto del servidor, los clientes necesitan el tipo del objeto. Para ello, se hace uso de las interfaces. De esta forma, conoce el tipo y los métodos que tiene el objeto pero no conoce su implementación.

Por ejemplo:



O1 y O2 están conectados a O y el coste computacional de ejecutar un método recae sobre el servidor. El único que hace un new es el servidor, los clientes no lo hacen porque estarían creando un nuevo objeto.

La estructura de una aplicación con RMI es la siguiente:



Interfaz del recurso

```
public interface IRecurso extends Remote{

    int incrementa(String quien, int valor) throws RemoteException;

}
```

La interfaz del recurso debe heredar de Remote y **todos** los métodos deben lanzar la excepción RemoteException. La interfaz del recurso debe ser la misma tanto en la aplicación cliente como en la aplicación servidor.

Implementación de la interfaz del recurso

```
public class Recurso extends UnicastRemoteObject implements IRecurso
{
    private int contador = 0;

    public Recurso() throws RemoteException{
        super();
    }

    @Override
    public synchronized int incrementa(String quien, int valor)
    throws RemoteException{

        contador += valor;
        System.out.println(quien + " incrementa en " + valor + " y
        lo deja en " + contador);
        return contador;
    }
}
```

La clase del recurso debe heredar de UnicastRemoteObject e implementar los métodos de la interfaz del recurso. Debemos tener en cuenta:

- El constructor del recurso debe llamar a super(); para invocar el constructor de UnicastRemoteObject y el constructor de Remote de la interfaz del recurso. Como consecuencia del constructor de Remote, se lanza la excepción RemoteException.
- Los métodos deben tener la cláusula Override.
- Los métodos que accedan a las variables del recurso deben estar sincronizados para controlar la concurrencia.
- Los métodos deben lanzar la excepción RemoteException para implementar correctamente la interfaz del recurso.

Implementación del servidor

```
public class Servidor {

    public static void main(String[] args) throws RemoteException,
    {
        Recurso r = new Recurso();

        Registry reg = LocateRegistry.createRegistry(2019);

        reg.rebind("Rremoto", r);

    }
}
```

Un objeto Registry es un objeto que se asocia a un puerto del servidor. En un registro podemos almacenar objetos de la misma clase y objetos de clases diferentes. Los objetos que estén en el registro serán los que se hagan públicos.

- `LocateRegistry.createRegistry(2019);` Esta sentencia asocia el objeto Registry a un puerto del ordenador.
- `reg.rebind("Rremoto", r);` Esta sentencia introduce un objeto en el registro y lo hace público. El nombre que se le pasa como primer parámetro lo debe conocer el cliente para poder conectarse a ese objeto. Si el objeto ya existía en el registro, el método sobrescribe el objeto antiguo por el que se le pasa por parámetro y si no existía lo introduce sin sustituir a ningún otro.

El servidor se ejecuta antes que las aplicaciones cliente, ya que sino los clientes no tienen a dónde conectarse. Aunque veamos que en el main el servidor ha terminado sus sentencias, la aplicación servidor no ha finalizado porque el servidor sigue escuchando en el puerto 2019. Si queremos que el servidor termine cuando pulsemos una tecla, añadimos las siguientes sentencias.

```
public class Servidor {

    public static void main(String[] args) throws RemoteException,
    IOException {
        Recurso r = new Recurso();

        Registry reg = LocateRegistry.createRegistry(2019);

        reg.rebind("Rremoto", r);

        System.out.println("Pulsa intro para parar");
        System.in.read();
        System.exit(0);

    }
}
```

Programación Concurrente y Distribuida (PCD)

Implementación del cliente

```
public class Cliente {

    public static void main(String[] args) throws NotBoundException,
        MalformedURLException, RemoteException {

        IRecurso r = (IRecurso) Naming.lookup("rmi://127.0.0.1:2019/Rremoto");

        System.out.println("Le sumo 8 y queda en " + r.incrementa("Raúl",8));

    }

}
```

Nota: Los ejemplos que se muestran tienen omitidas las sentencias de import.

Naming.lookup("rmi://127.0.0.1:2019/Rremoto"); Este método permite conectarnos a un objeto del servidor. Se pasa por parámetro una URL que se compone de:

- Protocolo: el protocolo que usamos es java rmi pero podría haber sido TCP, UDP... para otro tipo de aplicaciones. Cuando trabajemos con rmi el protocolo será rmi.
- IP: una dirección IP a la que conectarnos. En este caso hemos usado la de loopback.
- Puerto: el puerto en el que escucha la aplicación servidor.
- Nombre del recurso: el nombre con el que el recurso se publicó en el registro.

Podemos ver todos los objetos que hay publicados en un registro desde la aplicación cliente añadiendo las siguientes sentencias:

```
public class Cliente {

    public static void main(String[] args) throws NotBoundException,
        MalformedURLException, RemoteException {

        Registry Registro = LocateRegistry.getRegistry("127.0.0.1", 2019);
        //Vemos lo que ofrece el registro
        String[] oferta = Registro.list();

        for (int i = 0; i < oferta.length; i++) {
            System.out.println("Elemento " + i + " del registro: " + oferta[i]);
        }

        IRecurso r = (IRecurso) Naming.lookup("rmi://127.0.0.1:2019/Rremoto");

        System.out.println("Le sumo 8 y queda en " + r.incrementa("Raúl",8));

    }

}
```

Formación
Online
Especializada

Clases Online
Prácticas
Becas

Ponle
nombre
a lo que
quieres ser

Jose María Girela
Bim Manager.

