

El contenido de cada tema que va tras las líneas hechas con guiones son información extraída de los documentos del campus, la que va antes de esa línea son contenido de las diapos, el texto en cursiva son respuestas de preguntas de teoría o bastante core de memorizar.

# Tema 1

## Bases de la Conurrencia

**Un proceso** es un programa que puede tener varios hilos (secuencia de instrucciones, que se ejecutan de forma independiente).

**Los hilos no comparten** la pila ni el contador de programa.

**Comparten memoria de proceso y los recursos del sistema.**

Concurrencia vs Paralelismo

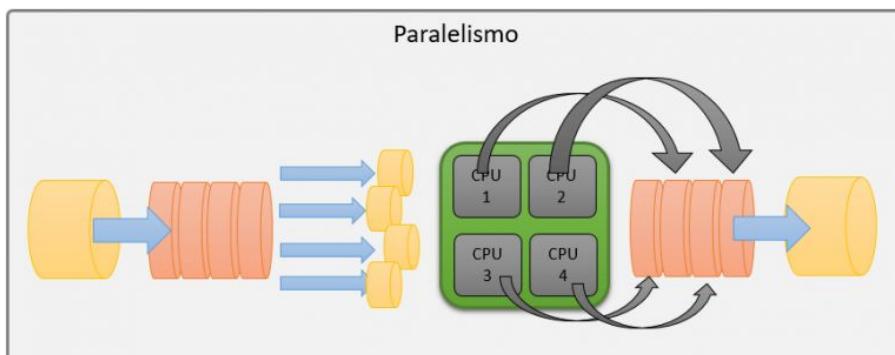
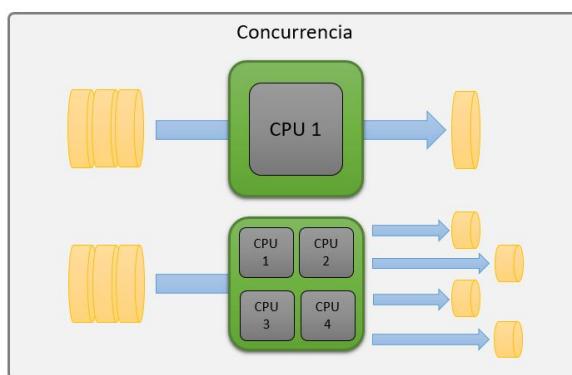
Fuente: <https://www.oscarblancarteblog.com/2017/03/29/concurrencia-vs-paralelismo/>

**La concurrencia** es la capacidad del CPU para procesar más de un proceso al mismo tiempo mientras que el **paralelismo** sigue la filosofía de "divide y vencerás", ya que consiste en **tomar un único problema, y mediante concurrencia llegar a una solución más rápido**. Mejora el rendimiento. El paralelismo lo que hace es tomar el problema inicial, dividir el problema en fracciones más pequeñas, y luego cada fracción es procesada de forma concurrente, aprovechando al máximo la capacidad del procesador para resolver el problema.

Ejemplo en la vida real:

**Concurrencia:** Descargar varias canciones a la vez.

**Paralelismo:** Página de vuelos que busca qué aerolíneas son más baratas.



**El contenido de cada tema que va tras las líneas hechas con guiones son información extraída de los documentos del campus, la que va antes de esa línea son contenido de las diapos, el texto en cursiva son respuestas de preguntas de teoría o bastante core de memorizar.**

Sistemas **inherentemente concurrentes**: el entorno con el que interactúan, o el entorno que modelan tiene **forzosamente actividades simultáneas** p.ej. red de cajeros automáticos.

Sistemas **potencialmente concurrentes**: no es estrictamente necesario que haya concurrencia, pero se puede sacar partido de ella p.ej. para aumentar la velocidad de ejecución.

**La atomicidad** es la propiedad que asegura que una operación se ha realizado o no, y por lo tanto ante un fallo del sistema no puede quedar a medias. Se dice que una operación es **atómica** cuando es imposible para otra parte de un sistema encontrar pasos intermedios. Si esta operación consiste en una serie de pasos, todos ellos ocurren o ninguno. Por ejemplo, en el caso de una transacción bancaria o se ejecuta tanto el depósito y la deducción o ninguna acción es realizada.

**Speedup:** coeficiente entre el tiempo necesario para completar una tarea secuencialmente y el tiempo necesario para hacerlo en paralelo. A partir de este coeficiente podremos determinar si es conveniente emplear la solución paralela o no.

$$S = \text{tiempo en secuencial} / \text{tiempo en paralelo}$$

1. Si  $S < 1$ , la paralelización ha hecho que la solución empeore. En este caso, mejor no paralelizar.
2. Si  $1 < S \leq$  número de núcleos, hemos conseguido mejorar la solución en un rango normal. En tareas puramente de cómputo (coeficiente de bloqueo muy cercano a 0), S debe estar muy próximo al citado número de núcleos.
3. Si  $S >$  número de núcleos, la mejora obtenida ha sido hiperlineal. Son casos muy poco comunes y solo se dan para determinados problemas ejecutándose en determinadas arquitecturas de memoria caché.

**Exclusión mutua:** Consiste en evitar la condición de concurso sobre un recurso compartido forzando la ejecución atómica de las secciones críticas de las entidades concurrentes que lo usan. Esto implica que haya **sincronización** (detener la ejecución de una entidad concurrente hasta que se produzcan determinadas circunstancias) y el poder comunicar a otras entidades el estado de una dada (comunicación). Usa protocolos de entrada y salida para acotar el código. Ésta nos garantiza que solo un hilo entra en la S.C. y también condiciones de vivacidad: garantiza que varios hilos pueden acceder a la misma sección tantas veces como quiera sin que ningún otro hilo lo impida.

#### **Beneficios:**

- Dentro de las secciones críticas no hay entrelazado.
- Se incrementa el determinismo, ya que se garantiza la ejecución secuencial ("atómica") de las secciones críticas.
- Permite comunicar a los procesos a través de las secciones críticas.
- Acota a nivel de código (sintácticamente) las secciones críticas mediante el uso de protocolos de entrada y salida de las mismas.
- Pueden generar sobrecargas de ejecución.

**Ejecución atómica:** Ejecución secuencial.

**El contenido de cada tema que va tras las líneas hechas con guiones son información extraída de los documentos del campus, la que va antes de esa línea son contenido de las diapos, el texto en cursiva son respuestas de preguntas de teoría o bastante core de memorizar.**

## Jerarquía de Flynn

		Datos	
		Simples	Múltiples
Instrucciones	Simples	SISD	SIMD
	Múltiples	MISD	MIMD

**SISD:** se refiere a una arquitectura computacional en la que un **único procesador** ejecuta **un sólo flujo de instrucciones**, para operar sobre datos almacenados en una **única memoria**.

Ej: UNIVAC 1, CRY 1, IBM360

### Características:

- Por cada ciclo de reloj se procesa un único dato y una única instrucción
- Ejecución determinista.
- El modelo más antiguo y el más extendido.

**MISD:** muchas unidades funcionales realizan diferentes operaciones en los **mismos datos**. (múltiples instrucciones para un mismo dato).

Ej: No existen muchos ejemplos de esta arquitectura, ya que MIMD y SIMD son a menudo más apropiados para técnicas comunes de datos paralelos. Sin embargo un ejemplo de prominente de la MISD son los controladores de vuelo del transbordador espacial.

### Características:

- Cada unidad ejecuta una instrucción distinta. (diferentes instrucciones)
- Cada unidad procesa el mismo dato. (un dato)
- Aplicación limitada en la vida real.

**SIMD:** Permite efectuar **varias operaciones** de cálculo con una **sola instrucción**. Nace debido a la necesidad de aplicar repetidamente una misma operación en grupos de datos diferentes. A los procesadores basados en esta arquitectura, se les conoce como procesadores matriciales. Utilizan memoria distribuida. Se utilizan en redes neuronales.

Ej: procesamiento de imágenes, redes neuronales, ILLIAC IV, CRY Y -MP, Thinkin Machine, GPU.

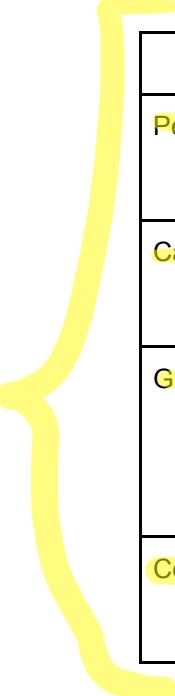
### Características:

- Todas las unidades ejecutan la misma instrucción.
- Cada unidad procesa un dato distinto.
- Todas las unidades operan simultáneamente.

**MIMD:** Es un sistema con un flujo de **múltiples instrucciones** que operan sobre **múltiples datos**. Se puede decir que es un superconjunto de SIMD. Las máquinas que usan MIMD tienen un número de procesadores que funcionan de manera asíncrona e independiente. Las computadoras MIMD pueden categorizarse por tener memoria compartida o distribuida. Ejecución (no)/ determinista, (a)síncrona.

Ej: Cluster IBM Power 5, Cluster Cray XT3, Cluster AMD Opteron, Procesador multicore Xeon 5600

El contenido de cada tema que va tras las líneas hechas con guiones son información extraída de los documentos del campus, la que va antes de esa línea son contenido de las diapos, el texto en cursiva son respuestas de preguntas de teoría o bastante core de memorizar.



Criterio de comparación	Multicore	Manycore
Perspectiva de crecimiento	Se estima que este tipo de procesadores <b>no tendrán una gran evolución</b>	Se estima que <b>la evolución</b> de los computadores <b>seguirá esta vía</b> .
Cantidad de núcleos	Entre 2 y 16	Cantidad variable, pero a nivel comercial <b>existen</b> arquitecturas con <b>más de 480 núcleos</b> .
Grado de paralelismo	Posee <b>paralelismo</b> , pero en un grado menor a las arquitecturas Manycore, debido a su acotada cantidad de núcleos.	Entrega <b>un alto grado de paralelismo</b> , debido a la gran cantidad de núcleos que posee.
Complejidad de los núcleos	Posee <b>núcleos altamente complejos</b> , de gran capacidad y tamaño.	Posee <b>núcleos simples</b> y de capacidades acotadas.

**Atomic Statements :** Instrucciones que se ejecutarán y finalizarán sin la posibilidad de entrelazado de instrucciones procedentes de otro proceso. Esto proporciona la capacidad de ejecutar dos procesos concurrentes y obtener el mismo resultado que la ejecución secuencial de estos, por lo tanto no habrá inconsistencias.

**Atomic Access :** Para el uso de acciones atómicas en la especificación de la API estándar Java (NO la de alto nivel y otras) se puede especificar que son atómicas las operaciones de lectura y escritura de las variables por referencia y por la mayoría de tipos primitivos de variables (excepto long y double), además **las operaciones de lectura y escritura de las variables indicadas como Volatile NO son atómicas, aunque sí lo son las de CARGA y GUARDADO**. El uso de alguna de las dos opciones NO elimina por completo la necesidad de sincronizar las acciones.

API Link : <https://docs.oracle.com/javase/tutorial/essential/concurrency/atomic.html>

**Volatile and Non-Atomic Variables :** Volatile es una palabra reservada de Java que **sirve para dar atomicidad a las operaciones de escritura y lectura sobre una variable** (en caso de un Array la referencia al mismo será volatile mientras que los elementos no lo serán) Volatile (Ej.: una variable static de una clase

El contenido de cada tema que va tras las líneas hechas con guiones son información extraída de los documentos del campus, la que va antes de esa línea son contenido de las diapos, el texto en cursiva son respuestas de preguntas de teoría o bastante core de memorizar.

public static volatile int contadorAtomico = 0;

), en la API de oracle explica que volatile REDUCE el riesgo de errores de consistencia en memoria.

Si una variable es marcada como volatile, cada vez que la variable es usada, el valor debe ser leído de la memoria principal. De manera similar, cada vez que la variable sea escrita, el valor debe ser almacenado en la memoria principal.

Volatile elimina las cachés temporales de las variables asociadas al motor de ejecución de las hebras que operan con esa variable.

*They can be used only when the operations that use the variable are atomic, meaning the methods that access the variable must use only a single load or store. If the method has other code, that code may not depend on the variable changing its value during its operation. For example, operations like increment and decrement ( ++, -- ) can't be used on a volatile variable because these operations are syntactic sugar for load, change and a store.*

**Entrelazado** : Dada una línea de tiempo, el entrelazado es la característica que denota diferentes grupos de instrucciones de distintos procesos en la CPU lo cual se produce por diferentes ejecuciones de distintos hilos. Es una característica que se produce en los sistemas concurrentes y paralelos. Ej :

### Ejemplo de Entrelazado: Efectos Laterales Indeseables

PROC.	INST.	N	REG1	REG2
Inicial		0		
P1	Load(x)	0	0	
P2	Load(x)	0	0	0
P1	Add(x, 1)	0	1	0
P2	Add(x, 1)	0	1	1
P1	Store(x)	1	1	1
P2	Store(x)	1	1	1

**Corrección** : Característica que denota la posibilidad de siempre obtener la misma salida a partir de la misma entrada, esta característica está innata en los programas secuenciales (determinismo  $\Leftrightarrow$  orden total). En los programas concurrentes no son deterministas ya que la misma entrada puede dar diferentes salidas (Indeterminista  $\Leftrightarrow$  orden parcial), esto no significa que sea Incorrecto,

**El contenido de cada tema que va tras las líneas hechas con guiones son información extraída de los documentos del campus, la que va antes de esa línea son contenido de las diapos, el texto en cursiva son respuestas de preguntas de teoría o bastante core de memorizar.**

### **Propiedades de corrección :**

Las propiedades de corrección se definen en términos de propiedades de sus secuencias de ejecución.

" Para todas las posibles ejecuciones de P.....", siendo P un hilo

- **Seguridad:** No debe ejecutar algo que conduzca a un error, no se da cuando tiene :
  - Ejecuta una operación imposible
  - **Entra en una sección crítica cuando está otro proceso.**
  - Un proceso no respeta un punto de sincronismo.
  - Un proceso permanece a la espera de un evento que nunca se producirá.
- **Vivacidad:** Las sentencias que se ejecuten deben contribuir a un avance constructivo al objetivo del programa, no se da cuando tiene :
  - Bloqueos activos: Dos procesos ejecutan sentencias que no hacen avanzar al programa
  - Aplazamiento indefinido: Un programa se queda sin tiempo de procesador para avanzar
  - Que no haya interbloqueos
- **Inanición:** Ninguna hebra puede quedarse indefinidamente sin poder progresar y acceder a la región crítica.
- **Equidad:** todas las hebras deben tener la misma probabilidad de progresar y acceder a la región crítica.

En los **programas secuenciales hay orden total** de la ejecución de las líneas de código, es decir, ante un conjunto de datos de entrada, el programa obtendrá el mismo resultado por muchas veces que se ejecute (determinista).

En cambio, en los **programas concurrentes hay un orden parcial** ya que ante el mismo conjunto de datos de entrada no se puede saber cuál va a ser el flujo de ejecución. En cada ejecución del programa el flujo puede ir por distinto sitio.

### **Explicación de propiedades de corrección:**

[https://www.ctr.unican.es/asignaturas/procodis\\_3\\_II/Doc/Procodis\\_1\\_04.pdf](https://www.ctr.unican.es/asignaturas/procodis_3_II/Doc/Procodis_1_04.pdf)

**Corrección parcial** [Si el programa termina, realiza su función correctamente. Garantiza el acceso en exclusión mutua a la sección crítica] + **terminación** (el programa termina alguna vez) = **Corrección total**

**Programa correcto parcialmente:** Cuando se garantizan la propiedad de seguridad, es decir, cuando no se accede de forma concurrente a variables compartidas. Osea, no se modifican variables compartidas al mismo tiempo por más de un hilo.

**Programa correcto totalmente:** Si además de ser correcto parcialmente no se bloquea.

El contenido de cada tema que va tras las líneas hechas con guiones son información extraída de los documentos del campus, la que va antes de esa línea son contenido de las diapos, el texto en cursiva son respuestas de preguntas de teoría o bastante core de memorizar.

## TEMA 3

# Creación y control de Threads en Java

Los hilos de un mismo proceso comparten el espacio de direcciones virtuales y los recursos del sistema pero poseen su propio contador de programa y pila de llamadas a procedimientos.

Para crear un hilo, si extiende de Thread ->

```
Clase h1 = new Clase ( a1,a2,a3 );
h1.start()
h2.join()
```

si es implements Runnable ->

```
Clase h1 = new Clase ( a1,a2,a3);
Thread hilo = new Thread (h1)
hilo.start()
hilo.join()
```

Se usa más el Runnable porque en Java no hay herencia múltiple. Si usamos Runnable podemos hacer que herede de otra clase mientras que con Thread no.

Con el método **getName()** podemos saber el id del hilo.

El método **join ()** hace que el hilo que actualmente está en ejecución pare hasta que el hilo especificado complete su tarea. Lo que hace el join es esperar a que el hilo termine de ejecutar el método run()

El hilo padre es la clase en sí que hereda de Thread o Runnable y los hilos hijos son los hilos que se crean dentro de esta.

La excepción que puede dar es **InterruptedException**.

El método **yield()** sugiere que se pueden ejecutar otras hebras de la MISMA PRIORIDAD. Si no hay hilos esperando, el hilo actual se ejecutará.

**Prioridad:** Para ver qué hilo se ejecuta antes. Indica al planificador la importancia de la hebra. -\* void **setPriority(int p)** y int **getPriority ()**

Ejemplo:

```
h1.setPriority(10);
h1.setPriority(1);
```

**Latencia:** suma de retardos temporales dentro de una red.

Una forma de ver la prioridad de un hilo sería poner Thread.currentThread().getPriority().

El pool de Threads se crea una vez y va reutilizando los hilos. Las tareas a ejecutar las recibe mediante objetos Runnable.

**Procesan tareas, no hebras.**

**El contenido de cada tema que va tras las líneas hechas con guiones son información extraída de los documentos del campus, la que va antes de esa línea son contenido de las diapos, el texto en cursiva son respuestas de preguntas de teoría o bastante core de memorizar.**

Permiten un mejor diseño del programa. Si nuestro programa tiene muchas tareas que ejecutar podemos escoger entre encargarnos de la gestión de los hilos o delegar en el ejecutor para la gestión de hilos y centrarnos más en la lógica del programa, esta segunda opción conlleva a programas más elegantes.

Cómo se crean:

```
#import java.util.concurrent.*;
```

Si tengo la clase Tarea:

**ExecutorService ejecutor = Executors.newSingleThreadExecutor();** -> Solo crea un hilo.

-> Serializa las tareas y mantiene su propia cola (oculta) de tareas pendientes

**ExecutorService ejecutor = Executors.newFixedThreadPool(500);** -> Ejecutor tamaño fijo.

**ExecutorService ejecutor = Executors.newCachedThreadPool();** -> Ejecutor tamaño variable.

-> Crea más hebras si las necesita.

```
for(int i = 0; i < 1000; i++)
    ejecutor.execute(new Tarea());
```

**ejecutor.shutdown();** -> impide que se envíen nuevas tareas a ese objeto Executor

**ejecutor.awaitTermination(1, TimeUnit.DAYS);** o poner debajo **while(!ejecutor.isTerminated());**

**Situaciones y ejecutores a utilizar:**

-Imagine que tiene una interfaz gráfica que realiza los cálculos sobre un autómata 2D, y que en un momento dado se le exige detener la ejecución de todas las hebras cuando el usuario pulse un botón 'STOP', pues para esto creamos SingleThreadExecutor que llame a Thread.interrupt() de los hilos que están procesando el autómata.

-Cuando le piden en el examen de prácticas utilizar un ejecutor acorde al número de núcleos de la máquina, se utiliza "int numeroCores = Runtime.getRuntime().availableProcessors();"

```
ejecutor = Executors.newFixedThreadPool(numeroCores);
```

**Ejecutores altamente configurables**

Cómo se crean:

```
ThreadPoolExecutor nombrePool = new ThreadPoolExecutor ( int corePoolSize , int
maximumPoolSize , long keepAliveTime , TimeUnit unit , BlockingQueue < Runnable > workQueue );
ThreadPoolExecutor miPool = new ThreadPoolExecutor ( tamPool , tamPool , 60000L ,
TimeUnit .MILLISECONDS , new LinkedBlockingQueue < Runnable > () );
Ej:
```

```
Tarea [] tareas = new Tarea [ nTareas ];
```

```
for ( int i = 0; i < nTareas ; i ++ ) {
    tareas [i] = new Tarea (i);
    miPool . execute ( tareas [i]);
}
```

**El contenido de cada tema que va tras las líneas hechas con guiones son información extraída de los documentos del campus, la que va antes de esa línea son contenido de las diapos, el texto en cursiva son respuestas de preguntas de teoría o bastante core de memorizar.**

miPool.shutdown();

---

**Coefficiente de bloqueo :**

**Latencia de red :**

**Why Thread Pools?** : Los motivos de uso de Pools serían principalmente :

- **El coste general de la creación de hilos es bastante alto, mediante el uso de Pools podemos conseguir cierta mejora cuando los hilos son reutilizados,** el grado de mejora es dependiente del programa y sus requisitos.
- **Los Pools tienen importantes beneficios para el rendimiento de las aplicaciones que utilizan muchos hilos simultáneamente.** En las veces que tenemos más hilos activos que cores un Pool puede dar más rapidez y eficiencia al programa desarrollado.

**Why Not Thread Pools** : Veamos estos casos :

- **El programa está haciendo procesamiento por lotes** (ejecución de un programa sin el control o supervisión directa del usuario).
- **El programa proporciona una sola respuesta de interés**, es decir, si no hay interés en los resultados dados por cada hilo no importa si uno termina antes que otro.
- **No son necesarios cuando las CPUs disponibles son las adecuadas para manejar todo el trabajo que el programa necesita.** en este caso un pool sería más dañino que beneficioso.

**Diferencia entre shutdown() y shutdownNow()** : El método `shutdown()` termina de forma "correcta" el ejecutor, es decir, las tareas que HAYAN SIDO ENVIADAS (con el `execute()`) serán las únicas que se les permitirá ejecutarse, pero otras nuevas tareas no serán aceptadas, el ejecutor se detendrá cuando todas las tareas aceptadas hayan terminado.

Por otra parte el método `shutdownNow()` procurará terminar de detener la ejecución del ejecutor en menos tiempo, las tareas que estén en EJECUCIÓN se interrumpen y es posible comprobar su estado ("*Still, existing tasks continue to run: they are interrupted, but it's up to the runnable object to check its interrupt status and exit when convenient.*"

Las tareas que no estén en ejecución pero sí han sido enviadas al ejecutor no serán procesadas y se devolverán en una lista ("all tasks that have not yet started are not run and are instead returned in a list").

**Nota Adicional** : En el documento del Tema 3 Gestión de la Concurrencia con Pool de Threads explica el uso combinado de Pools y la interfaz Callable aplicada con Future. En este archivo viene luego explicado.

El contenido de cada tema que va tras las líneas hechas con guiones son información extraída de los documentos del campus, la que va antes de esa línea son contenido de las diapos, el texto en cursiva son respuestas de preguntas de teoría o bastante core de memorizar.

## TEMA 4

# Modelos teóricos de control de la concurrencia

**La sección crítica** es la parte del programa o segmento de código con un comienzo y un final claramente marcados que permiten la modificación de recursos (variables) compartidos por diversos procesos.

**Una región crítica** es una sección crítica cuya ejecución bajo exclusión mutua está garantizada. (Java alto nivel)

**Mecanismo de espera ocupada:** es hacer un bucle hasta que se cumpla determinada condición.

**Inconvenientes región crítica:**

- Puede producir interbloqueos
- No da soporte para resolver sincronización.
- En C/C++ no existen como tales, aunque se pueden simular con otras primitivas.

### PREGUNTA EXÁMEN

¿Es posible definir regiones críticas en c++?

En c++ no existe una primitiva como en java para definir regiones críticas. En java existe el synchronized, pero en c++ no existe. Se puede simular la región crítica usando otras primitivas.

La **exclusión mutua** se consigue mediante protocolos de entrada y salida para la sección crítica, de modo que solo 1 proceso puede estar en la sección crítica y los demás que quieran entrar deben esperar a que ese proceso salga.

**Inconvenientes** de la exclusión mutua:

Espera ocupada, se requiere ser cuidadosos, muy ligados a la máquina, no son transportables, no dan una interfaz al programador, poco estructurados y difíciles de entender para un número arbitrario de entidades concurrentes.

### Semáforos

Es un tipo de datos abstractos que solo admite como posibles valores enteros no negativos.

Si solo admite el 1 y el 0, son **semáforos binarios**, en caso contrario **semáforos generales**.

Siendo s, el semáforo,

**El contenido de cada tema que va tras las líneas hechas con guiones son información extraída de los documentos del campus, la que va antes de esa línea son contenido de las diapos, el texto en cursiva son respuestas de preguntas de teoría o bastante core de memorizar.**

**wait (s):** Decrementa el valor de s si éste es mayor que 0. Si s es igual a 0, el proceso se bloqueará en el semáforo.

**signal (s):** Desbloquea algún proceso bloqueado en s, y en el caso de que no haya ningún proceso bloqueado incrementa el valor de s.

**initial (s,v):** Inicializa el semáforo s al valor v. Solo se usa en pascal y  $v \geq 0$ .

wait (s):

if  $s > 0$  then

$s := s - 1$

else bloquear proceso;

signal (s)

if( hay procesos bloqueados) then

    desbloquear un proceso

else  $s := s + 1$

**Inconvenientes:** bajo nivel, no estructurados, balanceado incorrecto de operaciones, semántica poco ligada al contexto, dificultad a la hora de usarlo, mantenimiento complejo.

**Java no tiene semáforos en la API estándar pero sí en la de alto nivel(since v1.5).** Pero posee primitivas de control de la exclusión mutua mediante cerrojos.

**Barrera:** Es un punto del código que ninguna entidad concurrente sobrepasa hasta que todas hayan llegado a ella.

**Monitores:** es un mecanismo de alto nivel de Software para control de concurrencia que contiene una colección de datos y los procedimientos necesarios para realizar la asignación de un determinado recurso o grupo de recursos compartidos por varios procesos.

Son una estructura de un lenguaje de programación similar a los semáforos en cuanto a funcionalidad, pero son más simples de manejar y controlar, son más estructurados, es decir un tipo de dato abstracto que encapsula datos privados y proporciona métodos públicos (Monitor = Encapsulación).

Def de las diapos: Un monitor es una construcción sintáctica de un lenguaje de programación concurrente que encapsula un recurso crítico a gestionar en exclusión mutua, junto con los procedimientos que lo gestionan. Por tanto, se da una centralización de recursos y una estructuración de los datos

→ Características:

- Es un Módulo de software.
- Tipo Abstracto de Dato (TAD).
- Mecanismo de alto nivel (impuesto por el compilador).
- Estructura Fundamental de Sincronización.
- Exclusión mutua (impuesta por la estructura del Monitor): sólo un proceso puede acceder al Monitor en cada momento, cualquier otro que lo invoque debe esperar.
- Variables de datos locales sólo se acceden a través de los procedimientos del Monitor.
- Un proceso entra al monitor invocando uno de sus procedimientos.
- Disciplinas de señalización más utilizadas:
  - Señalar y salir: la entidad que señala abandona el monitor.
  - Señalar y seguir: la entidad que señala sigue dentro del monitor.

**El contenido de cada tema que va tras las líneas hechas con guiones son información extraída de los documentos del campus, la que va antes de esa línea son contenido de las diapos, el texto en cursiva son respuestas de preguntas de teoría o bastante core de memorizar.**

**Ventajas:** Mueve todo el código de sincronización a los monitores y el resto de código ya no tiene que tener primitiva de sincronización. No me tengo q preocupar por bloquear el acceso a la variable compartida ya que esto lo hace el monitor.

→ Estructura sintáctica:

```
type nombre-monitor=monitor  
declaraciones de variables  
  
procedure entry P1(...)  
begin ... end;  
...  
procedure entry PN(...)  
begin ... end;  
  
begin  
    codigo de inicializacion  
end;
```

→ Disciplina de Señalización:

Tipo	Carácter	Clase de Señal
SA	Señales automáticas	Señales implícitas, incluidas por el compilador. No hay que programar send.
SC	Señalar y continuar	Señal explícita no desplazante. El proceso señalador no sale.
SX	Señalar y salir	Señal explícita desplazante. El proceso señalador sale. Send debe ser la última instrucción del monitor.
SW	Señalar y esperar	Señal explícita desplazante. El proceso señalador sale y es enviado a la cola de entrada del monitor.
SU	Señalar con urgencia	Señal explícita desplazante. El proceso señalador sale y es enviado a una cola de procesos urgentes, prioritaria sobre la de entrada.

La política de señalización depende el lenguaje de programación, no de si lo haces con while o if

**Señalar y seguir:** bucle while. Cuando salen del wait-set comprueban una condición.

**Señalar y salir:** Una entidad decide no continuar con la ejecución del método del monitor. Notifica a otros hilos de que ya pueden usar el monitor.

Supone ejecutar todo el código desde el principio. Los hilos que estén en el wait-set intentarán entrar en el monitor.

Se consigue con un if

#### PREGUNTA EXÁMEN

¿Que política de señalización usar para que el código sea correcto?

Si es condicional, es señalar y salir.

Si hay un bucle while, da igual que señalización ya que en todos los casos se va a comprobar de nuevo la condición.

La mayoría usan señalar y continuar.

¿Bucle while con señalar y salir? Si

**El contenido de cada tema que va tras las líneas hechas con guiones son información extraída de los documentos del campus, la que va antes de esa línea son contenido de las diapos, el texto en cursiva son respuestas de preguntas de teoría o bastante core de memorizar.**

→ **Con respecto a la sincronización:**

- ◆ **La exclusión mutua se asegura por definición, por lo tanto, solo un proceso puede estar ejecutando acciones de un monitor en un momento dado, aunque varios procesos pueden en ese momento ejecutar acciones que nada tengan que ver con el monitor**
- ◆ **La sincronización condicionada:**
  - **cada proceso puede requerir una sincronización distinta, por lo que hay que programar cada caso. Para ello se usarán variables de condición:**
    - **se usan para hacer esperar a un proceso hasta que determinada condición sobre el estado del monitor se "anuncie". También para despertar a un proceso que estaba esperando por su causa.**

→ **Sobre las variables de condición:**

- ◆ **Instrucciones wait:**
  - **el proceso invocador de la acción que la contiene queda dormido y pasa a la cola FIFO asociada a la variable, en espera de ser despertado.**
- ◆ **Instrucciones signal:**
  - **si la cola de la señal está vacía, no pasa nada y la acción que la ejecuta sigue con su ejecución. Al terminar el monitor está disponible para otro proceso.**
  - **si la cola no está vacía:**
    - **el 1º proceso de la cola se despierta (pero no avanza)**
    - **se saca de la cola**
    - **el proceso que la ejecuta sigue con su ejecución hasta terminar el procedimiento.**

En cuanto al "wait" y al "signal" es parecido a los semáforos pero:

- **Sii no hay proceso dormido, "signal" no hace nada.**
- **El "wait" del monitor siempre espera a un "signal" posterior.**
- **El proceso que ejecuta el procedimiento "signal" de un monitor sigue ejecutando la acción del monitor antes de que el que dormía pueda avanzar hasta que termina o queda dormido.**

---

**Origen de la Solución :**

Supongamos N procesos que tienen una secuencia de instrucciones, estas pueden ser divididas en dos subsecuencias, la sección crítica y la sección no crítica, para conseguir una primitiva de exclusión mutua se tiene que tener en cuenta las sig. características :

1. **Las especificaciones de corrección son requeridas para cualquier solución que busca aportar e.m. a la sección crítica :**
  - 1.1. **Exclusión mutua : Las instrucciones de la sec. crítica de dos o más procesos no deberán ser entrelazadas.**
  - 1.2. **Libres de deadlock : Si algunos procesos están tratando de entrar en su sec. crítica entonces uno de ellos eventualmente lo conseguirá. Lo que conlleva a que no habrá deadlock.**
  - 1.3. **Libres de (propia) ansia(de entrar) : Si algún proceso trat de entrar en la sec. crít. entonces ese proceso eventualmente lo conseguirá.**
2. **Un mecanismo de sincronización que asegurará los requisitos de corrección mencionados. Consistirá en instrucciones adicionales antes y después de la s.crit. (lo que llamamos protocolos de entrada y salida). Ver la sig. imagen.**
3. **Los protocolos de entrada y salida pueden requerir variables locales o globales.**

El contenido de cada tema que va tras las líneas hechas con guiones son información extraída de los documentos del campus, la que va antes de esa línea son contenido de las diapos, el texto en cursiva son respuestas de preguntas de teoría o bastante core de memorizar.

Ejemplo : El algoritmo de Dekker es correcto por cumplir la especificación de corrección.

### Algorithm 3.1. Critical section problem

global variables	
P	q
local variables loop forever non-critical section preprotocol critical section postprotocol	local variables loop forever non-critical section preprotocol critical section postprotocol

### Comparativa entre Semáforos y Monitores :

The following table summarizes the differences between the monitor operations and the similarly-named semaphore operations:

Semaphore	Monitor
<code>wait</code> may or may not block	<code>waitC</code> always blocks
<code>signal</code> always has an effect	<code>signalC</code> has no effect if queue is empty
<code>signal</code> unblocks an arbitrary blocked process	<code>signalC</code> unblocks the process at the head of the queue
a process unblocked by <code>signal</code> can resume execution immediately	a process unblocked by <code>signalC</code> must wait for the signaling process to leave monitor

**Problema de los lectores y escritores :** Este problema lo desarrollaron en clase (documento de Monitores del Tema 4) y muestra la situación de un documento que intenta ser accedido por lectores y escritores, si la persona que desea acceder es :

- **Lector :** Procesos que necesitan excluir a los escritores pero no a otros lectores, ya que el doc a leer no va a ser modificado por ellos por lo que varios pueden acceder a la vez.
- **Escritor :** Procesos que necesitan excluir a los lectores Y a los otros escritores ya que solo puede haber uno modificando el documento, además no puede haber ningún lector viendo el doc a la vez que se modifica.

El contenido de cada tema que va tras las líneas hechas con guiones son información extraída de los documentos del campus, la que va antes de esa línea son contenido de las diapos, el texto en cursiva son respuestas de preguntas de teoría o bastante core de memorizar.

#### Algorithm 7.4. Readers and writers with a monitor

```

monitor RW
    integer readers ← 0
    integer writers ← 0
    condition OKtoRead, OKtoWrite

    operation StartRead
        if writers ≠ 0 or not empty(OKtoWrite)
            waitC(OKtoRead)
        readers ← readers + 1
        signalC(OKtoRead)

    operation EndRead
        readers ← readers - 1
        if readers = 0
            signalC(OKtoWrite)

    operation StartWrite
        if writers ≠ 0 or readers ≠ 0
            waitC(OKtoWrite)
        writers ← writers + 1

    operation EndWrite
        writers ← writers - 1
        if empty(OKtoRead)
            then signalC(OKtoWrite)
            else signalC(OKtoRead)

```

reader	writer
p1: RW.StartRead p2: read the database p3: RW.EndRead	q1: RW.StartWrite q2: write to the database q3: RW.EndWrite

**Implementación de un semáforo con Java :** Teniendo en cuenta que es varios exámenes de teoría te piden implementar un semáforo o monitor es conveniente poner cómo lo hacen en el documento del campus, además lo hace lo más simple posible (aunque el volatile no es necesario, ver imagen) :

**El contenido de cada tema que va tras las líneas hechas con guiones son información extraída de los documentos del campus, la que va antes de esa línea son contenido de las diapos, el texto en cursiva son respuestas de preguntas de teoría o bastante core de memorizar.**

```
1 import java.util . concurrent. Semaphore;
2 class CountSem extends Thread {
3     static volatile int n = 0;
4     static Semaphore s = new Semaphore(1);
5
6     public void run() {
7         int temp;
8         for (int i = 0; i < 10; i ++) {
9             try {
10                 s. acquire ();
11             }
12             catch (InterruptedException e) {}
13             temp = n;
14             n = temp + 1;
15             s. release ();
16         }
17     }
18
19     public static void main(String[] args) {
20         // (as before)
21     }
22 }
```

El contenido de cada tema que va tras las líneas hechas con guiones son información extraída de los documentos del campus, la que va antes de esa línea son contenido de las diapos, el texto en cursiva son respuestas de preguntas de teoría o bastante core de memorizar.

## TEMA 5

# Control de la concurrencia en Java (API Estándar)

En Java, la exclusión mutua se logra a nivel de objetos. Cada uno tiene asociado un cerrojo. Los métodos sincronizados se encapsulan en una clase donde todos sus métodos modificadores son etiquetados con synchronized. Así pues 2 o más hilos están en exclusión mutua cuando llaman a métodos synchronized de un objeto o ejecutan métodos que tienen bloques de código sincronizados.

### ❖ Sintaxis bloque sincronizado:

```
public metodo_x ( parámetros )
{
    // Resto de código del método . No se ejecuta en EM.
    /* comienzo de la sección crítica */
    synchronized ( object )
    {
        // Bloque de sentencias críticas
        // Todo el código situado entre llaves se ejecuta bajo EM
    }
    /* fin de la sección crítica */
}
```

} // metodo\_x

### ❖ Sintaxis método sincronizado:

```
public void synchronized_ método_x (parámetros)
{
    // Bloque de sentencias del método
    // Todo el código del método se ejecuta en em-
}
```

} // método\_x

En clases heredadas, los métodos sobreescritos pueden ser sincronizados o no, sin afectar a cómo era (y es) el método en la superclase. Es decir, la sincronización no se hereda. Si deseamos que sean sincronizados debemos especificarlo en el código (sobreescribir el método).

Normalmente la razón para usar un bloque sincronizado en vez de un método sincronizado es para que las otras tareas puedan acceder más frecuentemente (siempre y cuando sea seguro hacerlo).

### ❖ Protocolos de Control de E.M en Java:

#### ➤ 1er Protocolo:

- Acceder al recurso compartido donde sea necesario.
- Definir un objeto para control de la exclusión mutua (o usar el propio objeto, this).
- Sincronizar el código crítico utilizando el cerrojo del objeto de control dentro de un bloque synchronized de instrucciones.

#### ➤ 2ndo Protocolo:

**El contenido de cada tema que va tras las líneas hechas con guiones son información extraída de los documentos del campus, la que va antes de esa línea son contenido de las diapos, el texto en cursiva son respuestas de preguntas de teoría o bastante core de memorizar.**

- Encapsular el recurso crítico en una clase
- Definir todos los métodos de la clase como no estáticos y synchronized, o al menos todos los modificadores.
- Crear hilos que comparten una instancia de la clase creada.

**Reentrantia:** Un método sincronizado puede invocar a otro método sincronizado sobre el mismo objeto.

Es posible llamar recursivamente a métodos sincronizados e invocar métodos heredados sincronizados. La reentrantia en Java es por defecto.

En java todas las primitivas son reentrantes por diseño, es decir, El bloqueo se produce a nivel de hilos, es decir, cuando haces synchronized( this) echa el cerrojo pero se queda con la llave. Por tanto puede pasar por ese cerrojo tantas veces como quiera.

Así que podemos hacer lo siguiente:

```
public synchronized void m1( )  
{  
    //System.out.println(ñlaksdc)  
}  
  
public synchronized void m2 ( )  
{  
    this.m1( );  
}
```

Esto no se bloquea en java por el motivo de antes. Pero si fuera otro lenguaje si se bloquería en el caso de que no fuera reentrant

En C++ las primitivas pueden ser reentrantes o no. Así que no podemos decir que C++ sea un lenguaje reentrant o no ya que depende las primitivas.

**Interbloqueos:** Se producen cuando hay condiciones de espera de liberación de bloqueos cruzados entre 2 hilos. Sólo pueden evitarse mediante un análisis cuidadoso y riguroso del uso de código sincronizado.

\*Mirar notas que puse en las diapos en la parte de los códigos

### Sincronización entre hilos:

En Java la sincronización entre hilos se logra con los métodos wait, notify y notifyAll(clase Object). Estos se usan únicamente dentro de métodos o código de tipo synchronized.

Cuando un hilo llama a un método que hace notify, uno de los hilos bloqueados en la cola pasa a listo. Java no especifica cuál. Depende de la implementación (JVM). Si se llama al método notifyAll, todos los hilos de dicha cola son desbloqueados y pasan a listo. Accederá al objeto aquél que sea planificado.

El contenido de cada tema que va tras las líneas hechas con guiones son información extraída de los documentos del campus, la que va antes de esa línea son contenido de las diapos, el texto en cursiva son respuestas de preguntas de teoría o bastante core de memorizar.

Las señales solo las reciben los procesos que lo esperan. Si no hay nadie en el wait set, la señal se pierde.

**Nota:** Planificador de procesos: es la parte del sistema operativo que se encarga de seleccionar a qué proceso se asigna el recurso procesador y durante cuánto tiempo.

Siempre que el hilo usuario **invoca a wait()**, lo primero al despertar es volver a comprobar su condición particular, volviendo al bloqueo si ésta aún no se cumple.

❖ Monitores:

- Implementa acceso bajo exclusión mutua a todos sus métodos y provee sincronización.
- En Java, son objetos de una clase cuyos métodos públicos son todos synchronized.
- Los métodos wait(), notify() y notifyAll() permiten sincronizar los accesos al monitor, de acuerdo a la semántica de los mismos ya conocida.
- Es conveniente usar notifyAll() para que todos los procesos comprueben la condición que los bloqueó ya que no hay variables de condición sólo una única variable implícita.
- Aconsejable bloquear los hilos con una condición de guarda junto al notifyAll()  
`while ( !condicion) try { wait(); } catch (InterruptedException e) {return lo q sea;}`
- No es aconsejable comprobar la condición de guarda con if
- Los campos protegidos por el monitor suelen declararse private.

### Técnica de Diseño de Monitores en Java

1. Decidir qué datos encapsular en el monitor
2. Construir un monitor teórico, utilizando tantas variables de condición como sean necesarias
3. Usar la señalización SC en el monitor teórico
4. Implementar en Java
  - 4.1 Escribir un método synchronized por cada procedimiento
  - 4.2 Hacer los datos encapsulados private
  - 4.3 Sustituir cada wait(variableCondicion) por una condición de guarda.
  - 4.4 Sustituir cada send(variableCondicion) por una llamada a notifyAll()
  - 4.5 Escribir el código de inicialización del monitor en el constructor del mismo

El contenido de cada tema que va tras las líneas hechas con guiones son información extraída de los documentos del campus, la que va antes de esa línea son contenido de las diapos, el texto en cursiva son respuestas de preguntas de teoría o bastante core de memorizar.

❖ Estructura:

```
1  class Monitor {  
2      //definir aquí datos protegidos por el monitor  
3      public Monitor() {...} //constructor  
4      public synchronized tipo1 metodo1()  
5          throws InterruptedException {  
6              ...  
7              notifyAll();  
8              ...  
9              while (!condicion1) wait();  
10         }  
11         public synchronized tipo2 metodo2()  
12             throws InterruptedException {  
13                 ...  
14                 notifyAll();  
15                 ...  
16                 while (!condicion1) wait();  
17             }  
18     }
```

❖ Semántica de un Monitor:

- Cuando un método **synchronized** del monitor llama a **wait()**, **libera la exclusión mutua** sobre el monitor y **encola al hilo** que llamó al método en el wait-set.
- Al hacer **notify()** el hilo pasa a la cola de hilos que esperan el cerrojo y se reanudará cuando sea planificado.
- El monitor Java no tiene **variables de condición**, sólo una cola de bloqueo de **espera implícita**.
- La política de señalización es SC. El método señalador sigue su ejecución y el hilo/s señalado/s pasan al wait-set a la cola de procesos que esperan el cerrojo.
- Para dormir a un hilo usar **wait()**

---

El texto de la documentación de este tema viene bien resumido y explicado en las diapos.

El contenido de cada tema que va tras las líneas hechas con guiones son información extraída de los documentos del campus, la que va antes de esa línea son contenido de las diapos, el texto en cursiva son respuestas de preguntas de teoría o bastante core de memorizar.

## TEMA 6

# Control de la concurrencia en Java (API de alto nivel)

No todos los métodos del API de estas clases soportan concurrencia segura. Sólo aquellos en los que se indica que la operación se realiza de forma atómica.

**Clases:** `AtomicBoolean`, `AtomicInteger`, `AtomicIntegerArray`, `AtomicLong`, `AtomicLongArray`, `AtomicReference`, etc.

Sobre esta clase se definen unos cuantos métodos con la garantía de que están `synchronized`.

Usar estas clases atómicas es mucho más lento que si no las usamos y en su lugar usamos atributos normales (`boolean`, `int`, etc..). Lo que pasa es que si usamos estos últimos, corremos el peligro de que se produzca entrelazado.

En los `Atomic`, para crear una variable e inicializarla:

`AtomicInteger cont = new AtomicInteger(0);`

Y para incrementar -> `this.cont.incrementAndGet();`

**Paquetes:**

- `java.util.concurrent`
- `java.util.concurrent.atomic`
- `java.util.concurrent.locks`

**Cosas que se usan:** Semáforos, Barreras, Variables atómicas, colecciones (clases contenedoras) concurrentes de acceso sincronizado.

### ❖ Clase `Semaphore`:

- Sus métodos principales son:
  - `Semaphore(long permits) //ctor`. Indica que x threads pueden acceder al recurso al mismo tiempo.
  - `acquire()`, `acquire(int permits)` //como el `wait`. Método no bloqueante. Solicita unos permisos. Si no hay suficientes, queda esperando. Si los hay, se descuentan del semáforo y se sigue. Si llega una interrupción, se aborta la espera.
  - `release()`, `release(int permits)` //como el `signal`. Libera unos permisos.
  - `tryAcquire()` //intentar adquirir el semáforo
  - `tryAcquire(long num, TimeUnit)` //bloquea el hilo el tiempo que se especifica
  - `availablePermits()`
- Para **protocolo de control de E.M** hacer en el programa ppal siempre  
`Semaphore s = new Semaphore(1)`
- Para **protocolo de Sincronización Inter-Hilos** hacer en el programa ppal siempre:  
`Semaphore s = new Semaphore(0)`

El contenido de cada tema que va tras las líneas hechas con guiones son información extraída de los documentos del campus, la que va antes de esa línea son contenido de las diapos, el texto en cursiva son respuestas de preguntas de teoría o bastante core de memorizar.

➤ Ejemplo:

```
public class Recurso {  
    //Indica que dos threads pueden acceder el recurso al mismo tiempo.  
    Semaphore semaphore = new Semaphore(2);  
    public void lock() {  
        //Si el contador es cero el thread se duerme, de lo contrario se reduce y obtiene el acceso |  
        semaphore.acquire();  
        System.out.println("Comenzando, el thread " + Thread.currentThread().getName() + " tiene el lock");  
        try {  
            Thread.sleep(10000);  
        } catch (InterruptedException e) {  
            e.printStackTrace();  
        }  
        System.out.println("Terminando, el thread " + Thread.currentThread().getName() + " tiene el lock");  
        semaphore.release(); //Libera el semaphore e incrementa el contador  
    }  
}
```

En el ejemplo, el método **acquire()** obtiene el semáforo si el contador es mayor que cero, de lo contrario se espera hasta que se incremente y reduce el contador, después el thread que obtiene el semáforo ejecuta el recurso y finalmente, se ejecuta el método **release()** y incrementa el contador.

Si el semáforo inicializa su contador con un valor de uno, entonces se llama un semáforo binario y se comporta como la interfaz `java.util.concurrent.locks.Lock`.

❖ Clase **CyclicBarrier**:

- Una barrera es un punto de espera a partir del cual todos los hilos se sincronizan.
- Ningún hilo pasa por la barrera hasta que todos los hilos esperados llegan a ella.
- Sirve para unificar resultados parciales o como inicio siguiente fase de ejecución simultánea.
- Como es cíclica, está se resetea una vez pasan todos los hilos
- Los primeros hilos se bloquearan y el último lo desbloqueara
- Desde el hilo padre puedo llamar al await (mirar código diapos 21)

❖ API **java.util.concurrent.locks**

- Proporciona clases para establecer sincronización de hilos alternativas a los bloques de código sincronizado y a los monitores.
- Clases:
  - ReentrantLock: proporciona cerrojos de e.m de semántica equivalente a synchronized, pero con un manejo más sencillo y una mejor granulidad (refiriéndose al tipo de hilo que queremos despertar. Con mejor detalle, que puedes desbloquear a un tipo de hilo u a otro).
  - LockSupport: proporciona primitivas de bloqueo de hilos que permiten al programador diseñar sus clases de sincronización y cerrojos propios.
  - Condition: es una interfaz cuyas instancias se usan asociadas a locks. Implementan variables de condición y proporcionan una alternativa de sincronización a los métodos wait, notify y notifyAll de la clase Object.

❖ API de la Clase **ReentrantLock (throws InterruptedException)**

- Proporciona cerrojos reentrantes de semántica equivalente a synchronized.
- Método **lock-unlock()**: duerme al actual hilo hasta que hayan soltado el candado si otro método ya ha adquirido el cerrojo.
- Método **isLocked()**

El contenido de cada tema que va tras las líneas hechas con guiones son información extraída de los documentos del campus, la que va antes de esa línea son contenido de las diapos, el texto en cursiva son respuestas de preguntas de teoría o bastante core de memorizar.

➤ Método **Condition newCondition()** que retorna una variable de condición asociada al cerrojo

➤ Protocolo de Control de E.M con **ReentrantLock**:

- Definir el recurso compartido donde sea necesario.
- Definir un objeto c de clase ReentrantLock para control de la exclusión mutua.
- Acotar la sección crítica con el par **c.lock() y c.unlock()**.

Ejemplo:

```
ReentrantLock lock = new ReentrantLock();
int count = 0;
```

```
void increment() {
    lock.lock();
    try {
        count++;
    } finally {
        lock.unlock();
    }
}
```

El candado se adquiere mediante **lock()** y se libera con **unlock()**.

Es importante envolver el código en un **try/finally** para asegurar el desbloqueo en caso de excepciones. Si otro método ya ha adquirido el candado, llama a lock() para dormir al actual hilo hasta que el candado lo hayan soltado. **Solo un hilo puede tener el candado a la vez.**

#### ❖ **Interfaz Condition**

- Se asocia a un cerrojo, siendo cerrojo una instancia de la clase ReentrantLock
- **cerrojo.newCondition()** retorna la variable de condición.
- Op. ppales: **await()**, **signal()** y **signalAll()**.

#### ❖ **Clases Contenedoras Sincronizadas**

- ConcurrentHashMap, ConcurrentSkipListMap, ConcurrentSkipListSet, CopyOnWriteArrayList y CopyOnWriteArraySet.

#### ❖ **Colas Sincronizadas**

- Clases: **LinkedBlockingQueue**, **ArrayBlockingQueue**, **SynchronousQueue**, **PriorityBlockingQueue** y **DelayQueue**
- Son seguras frente a hilos concurrentes.
- Proporcionan notificación a hilos según cambia su contenido.
- Son autosincronizadas. Pueden proveer sincronización (además de e.m) por sí mismas.
- Facilitan enormemente la programación de patrones de concurrencia como el P-S

## PREGUNTA EXÁMEN

Si en vez de usar un contenedor sincronizado, por ejemplo en vez de **LinkedBlockingQueue** usamos un array, ¿ Que sería más eficiente, usar la cola sincronizada o la no sincronizada?

La mas ineficiente seria usar la sincronizada pq al hacer put accederia de manera sincronizada. Pero si no lo uso, puedo pasar que el resultado no sea el esperado. No sería seguro

El contenido de cada tema que va tras las líneas hechas con guiones son información extraída de los documentos del campus, la que va antes de esa línea son contenido de las diapos, el texto en cursiva son respuestas de preguntas de teoría o bastante core de memorizar.

El texto de la documentación de este tema viene bien resumido y explicado en las diapos.

## TEMA 7

# Paso de mensajes en Java con RMI (Remote Method Invocation)

En la programación distribuida no existe memoria común, son escalables y reconfigurables.

### ❖ Modelos de Programación Distribuida:

#### ➤ **Modelo de paso de mensajes (op. send y receive)**

- Mecanismo para comunicar y sincronizar entidades concurrentes que no pueden o no quieren compartir memoria.
- Las llamadas pueden ser:
  - Bloqueadas-No bloqueadas
  - Almacenadas - No almacenadas
  - Fiables - No Fiables
- En UNIX, Pipes con y sin nombres
- En Java
  - Sockets: Clases Socket y ServerSocket
  - Canales CTJ (Communicating Threads for Java)

#### ➤ **Modelo RPC (Remote Procedure Call)**

- Es un programa que utiliza una computadora para ejecutar código en otra máquina remota sin tener que preocuparse por las comunicaciones entre ambas.
- Stubs de cliente y servidor
- Visión en Java es RMI (tipo particular de RPC. Cada lenguaje tendrá su propio RPC, en el caso de java se llama RMI)
- Llamadas a procedimiento local o remoto indistintamente

### ❖ **Introducción del Nivel de Stubs:**

- Los stubs son clases que se generan automáticamente para convertir esas llamadas a las funciones en secuencias de bytes
- Efectúan el marshalling/unmarshalling de parámetros  
*Marshalling | Unmarshalling* es el proceso por el cual debe pasar toda información para que ésta sea utilizable en ambientes heterogéneos. Si se ejecuta un programa en *una sola* computadora se tiene la seguridad que la representación de datos está conforme a esa plataforma pero ¿qué ocurre si se intentan pasar parámetros a un *procedimiento* en otra plataforma? Para eso está el marshalling/unmarshalling
- Bloquean al cliente a la espera del resultado.
- Proporcionan transparencia

El contenido de cada tema que va tras las líneas hechas con guiones son información extraída de los documentos del campus, la que va antes de esa línea son contenido de las diapos, el texto en cursiva son respuestas de preguntas de teoría o bastante core de memorizar.

- El Stub permite que al momento de ser invocada la función remota esta pueda ser "simulada localmente". Enmascara los detalles de comunicación entre los procesos. El stub es el proxy del cliente.

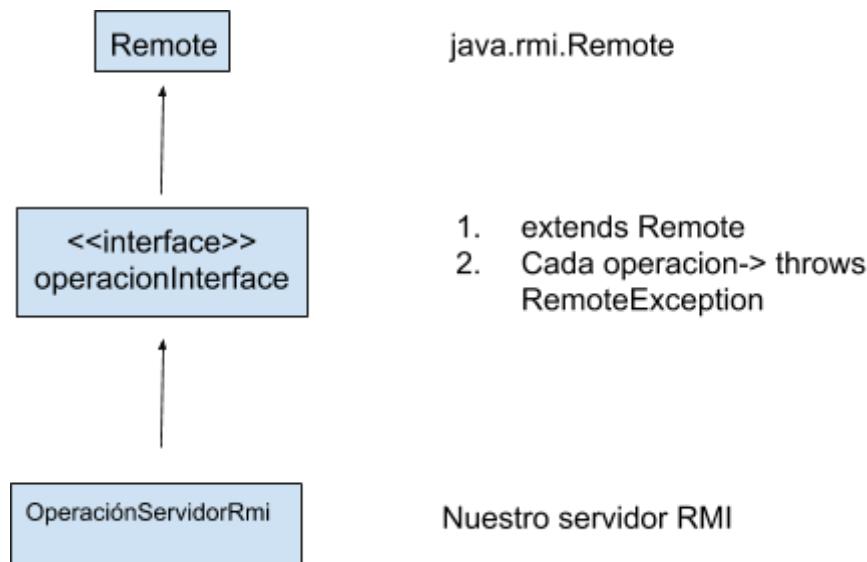
❖ **RMI en Java**

- Permite disponer de objetos distribuidos utilizando Java
- Un objeto distribuido se ejecuta en una JVM diferente o remota
- El objetivo es lograr una referencia al objeto remoto que permita utilizarlo como si el objeto se estuviera ejecutando sobre la JVM local.
- Similar a RPC pero con un nivel de abstracción más alto.
- Pasa los parámetros y valores de retorno utilizando serialización de objetos.

❖ **RPC vs RMI**

RPC	RMI
Carácter y Estructura de diseño procedimental	Carácter y Estructura de diseño orientada a objetos
Es dependiente del lenguaje	Es dependiente del lenguaje
Utiliza la representación externa de datos XDR	Utiliza la serialización de objetos en Java
El uso de punteros requiere el manejo explícito de los mismos	El uso de referencias a objetos locales y remotos es automático
NO hay movilidad de código	El código es móvil, mediante el uso de bytecodes.

El RMI habilita la prog. distribuida. ¿Para qué sirve? Es para que un objeto java pueda ofrecerse en una máquina y otros usuarios desde otras máquinas puedan usar el objeto y sus métodos desde otras máquinas. No necesito al objeto en mi máquina para poder usarlo.



El cliente y el servidor se van a comunicar a través de esa interfaz. Por lo que el cliente nunca va a saber exactamente quién es el servidor.

❖ **Llamadas Locales vs Llamadas Remotas**

El contenido de cada tema que va tras las líneas hechas con guiones son información extraída de los documentos del campus, la que va antes de esa línea son contenido de las diapos, el texto en cursiva son respuestas de preguntas de teoría o bastante core de memorizar.

- Un objeto remoto es aquel que se ejecuta en una JVM diferente, situada potencialmente en un host distinto.
- RMI es la acción de invocar a un método de la interfaz de un objeto remoto.

```

1 //ejemplo de llamada a metodo local
2 int dato;
3 dato = Suma (x,y);
4 //ejemplo de llamada a metodo remoto (no completo)
5 IEjemploRMI1 ORemoto =
6 (IEjemploRMI1)Naming.lookup("//sargo:2005/EjemploRMI1");
7 ORemoto.Suma(x,y);

```

#### ❖ Arquitectura RMI

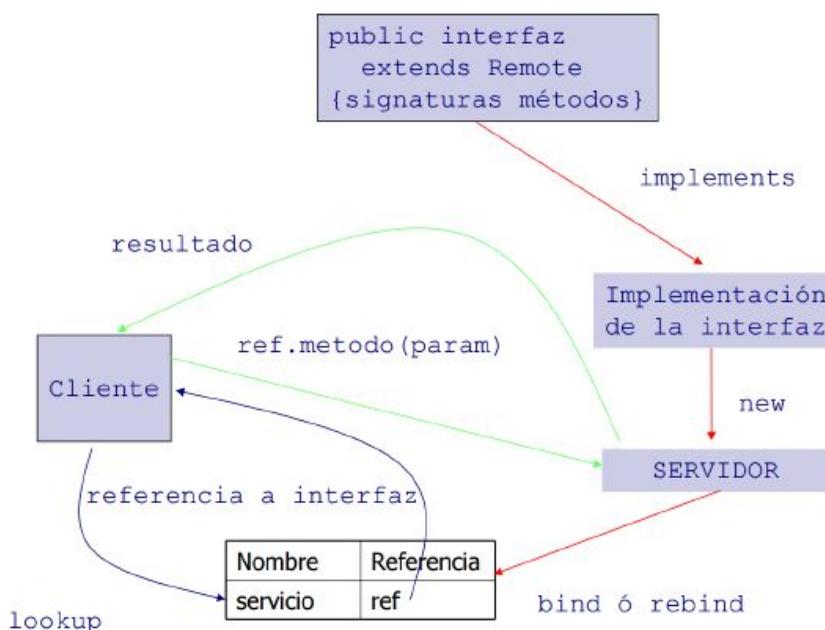
- El servidor debe extender **UnicastRemoteObject**
- El servidor debe implementar una interfaz diseñada previamente.
- El método main del servidor debe lanzar un gestor de seguridad
- El método **main crea los objetos remotos**
- El compilador de RMI (rmic) genera el stub y el skeleton. (El stub ya se genera dinámicamente por lo que esta instrucción está obsoleta)
- Los clientes de objetos remotos se comunican con **interfaces remotas** (diseñadas antes de...)
- Los objetos remotos son **pasados por referencia**.
- Los clientes que llaman a métodos remotos deben manejar excepciones.

#### ❖ Clase java.rmi.Naming

```

1 public static void bind(String name, Remote obj)
2 public static String[] list(String name)
3 public static Remote lookup(String name)
4 public static void rebind(String name, Remote obj)
5 public static void unbind(String name)

```



Dynamic-binding: Se enlaza en tiempo de ejecución

#### ❖ Fases de Diseño:

- Interface:

**El contenido de cada tema que va tras las líneas hechas con guiones son información extraída de los documentos del campus, la que va antes de esa línea son contenido de las diapos, el texto en cursiva son respuestas de preguntas de teoría o bastante core de memorizar.**

- Escribir el fichero de la interfaz remota.
- Debe ser **public** y extender a **Remote**.
- Declara todos los métodos que el servidor remoto ofrece, pero NO los implementa. Se indica nombre, parámetros y tipo de retorno.
- Todos los métodos de la interfaz remota lanzan obligatoriamente la excepción **RemoteException**
- El propósito de la interface es ocultar la implementación de los aspectos relativos a los métodos remotos.
- De esta forma, cuando el cliente logra una referencia a un objeto remoto, en realidad obtiene una referencia a una interfaz. Los clientes envían sus mensajes a los métodos de la interfaz.

➤ **Stub y Skeleton:**

- Es necesario que en la máquina remota donde se aloje el servidor se situen también los ficheros de stub y skeleton (proxy del servidor).
- Con todo ello disponible, se lanza el servidor llamado a JVM del host remoto, previo registro en un DNS.
- Se generan en tiempos de ejecución. Con la versión anterior de Java tenías que generarlos con rmic, pero ahora en las versiones modernas de java ya no.

➤ **Registro:**

- Método Naming.bind("Servidor",ORemoto). Para registrar el objeto remoto creado requiere que el servidor de nombres esté activo.
- Dicho servidor se activa con start rmiregistry en Win32
- Dicho servidor se activa con rmiregistry & en Unix.
- Se puede indicar como parámetro el puerto que escucha.
- Si no se indica, por defecto es el puerto 1099.
- El parámetro puede ser el nombre de un host como en Naming.bind("//sargo.uca.es:2005/Servidor",ORemoto);

➤ **Cliente:**

- El objeto cliente procede siempre creando un objeto de tipo **INTERFAZ** remota.
- Posteriormente efectúa una llamada al método Naming.lookup cuyo parámetro es el nombre y puerto del host remoto junto con el nombre del servidor.
- Naming.lookup devuelve **una referencia que se convierte a una referencia a la interfaz remota**.
- A partir de aquí, a través de esa referencia el programador puede invocar todos los métodos de esa interfaz remota como si fueran referencias a objetos en la JVM local.

❖ **Serialización**

- Serializar un objeto es convertirlo en una cadena de bits, posteriormente restaurada en el objeto original.
- Es útil para enviar objetos complejos en RMI.
- Los tipos primitivos y las clases contenedoras se serializan automáticamente
- Los objetos complejos deben implementar la interfaz Serializable para poder ser serializados.
- Java no siempre va a saber traducir cualquier objeto que le pasemos. ¿Qué tipo de datos podemos pasarle a los métodos que vamos a ejecutar en el servidor?  
Parámetros de tipo básico y atributos de cualquier otra clase siempre y cuando esta

**El contenido de cada tema que va tras las líneas hechas con guiones son información extraída de los documentos del campus, la que va antes de esa línea son contenido de las diapos, el texto en cursiva son respuestas de preguntas de teoría o bastante core de memorizar.**

clase sea serializable. Una clase es serializable cuando implementa la interfaz Serializable. En conclusión, solo podemos pasársela parámetros de tipo básico o serializables

❖ **Seguridad:**

- System.setSecurityManager(new RMISecurityManager());
- Se ajusta la seguridad construyendo un objeto SecurityManager y llamando al método setSecurityManager (clase System)
- Clase RMISecurityManager
- Programador ajusta la seguridad mediante la clase Policy
  - Policy.getPolicy() permite conocer la seguridad actual.
  - Policy.setPolicy() permite fijar nueva política
  - Seguridad reside en fichero específico.
  - El fichero sería fichero.policy y se pone en el directorio donde está el servidor y el cliente.
  - Al ejecutar el servidor o el cliente ponemos el comando:  
**java -Djava.security=fichero.policy servidor | cliente**
  - Java incorpora policytool, una herramienta visual

<http://laurel.datsi.fi.upm.es/~ssoo/SD.dir/practicas/guiarmi.html>

❖ **CallBack de Cliente:**

- Si el servidor de RMI debe notificar eventos a clientes, la arquitectura es inapropiada.
- La alternativa estándar es el sondeo (polling)
- Clientes interesados se registran en un objeto servidor para que les sea notificado un evento.
- Cuando el evento se produce, el objeto servidor notifica al cliente su ocurrencia.
- Callback es mucho más eficiente que el polling ya que no tiene que estar constantemente mirando cuál es el estado del sistema.
- Para realizar el callback, si el servidor quiere hacerle una petición/notificación al cliente debería saber quién es el cliente. Pero no podemos mandar desde el cliente un string con la IP del cliente porque ésta cambia dinámicamente. Lo que si podemos mandar desde el cliente es una referencia del mismo al servidor. Le mandamos una referencia del propio objeto cliente al servidor, lo que se conoce como registrar al cliente en el servidor

❖ **Descarga Dinámica de Clases:**

- Permite cambios libres en el servidor
- Elimina la necesidad de distribuir (resguardos) si los hay y nuevas clases.
- Dinámicamente, el cliente descarga todas las clases que necesita para desarrollar una RMI válida mediante HTTP o FTP

❖ **Compilación:**

- javac InterfazRemota.java
- javac Servidor.java
- rmic Servidor -> Compilador de RMI. Genera stub y skeleton (deprecated)
- **start rmiregistry** -> en windows **rmiregistry &** -> en ubuntu
  - Por defecto el registro se ejecuta en el puerto 1099. Para cambiar el puerto se hace de la siguiente forma:
    - start rmiregistry 2001
    - rmiregistry 2001 &
- java Servidor // se ejecuta el servidor
- javac Cliente.java

**El contenido de cada tema que va tras las líneas hechas con guiones son información extraída de los documentos del campus, la que va antes de esa línea son contenido de las diapos, el texto en cursiva son respuestas de preguntas de teoría o bastante core de memorizar.**

➤ java Cliente

Sondeo : Varias peticiones solicitadas por un cliente a un servidor que se repiten hasta obtener una respuesta deseada. Suponer que esperamos a que un server inicie una oferta :

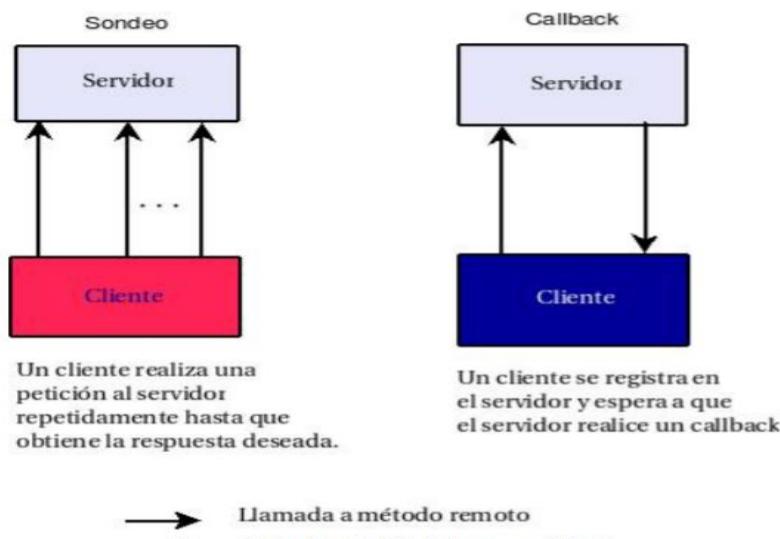
```
InterfazServidor h = (InterfazServidor) Naming.lookup(URLRegistro);
while (!(h.haComenzadoOferta())) {}
```

Callback del cliente(ampliación) : Característica que permite a un cliente de objeto registrarse a sí mismo con un servidor de objeto remoto para callbacks, de forma que el servidor puede llevar a cabo una invocación del método del cliente cuando un evento (previamente pedido por el cliente al servidor) ocurra y el servidor quiera notificar la finalización, esto conlleva a invocaciones de los métodos de forma bidireccional. Cuando se emplea el callback el servidor pasa a ser cliente y el cliente pasa a ser servidor. El cliente tendría que proporcionar una interfaz remota.

Otras características del uso de Callbacks :

- El servidor necesita emplear una estructura de datos que mantenga una lista de las referencias a la interfaz cliente registradas para callbacks. Cada registro implica añadir una referencia al vector.
- El uso (opcional) de políticas de seguridad de Java : Fichero de texto que contiene códigos que permiten especificar la concesión de permisos específicos.

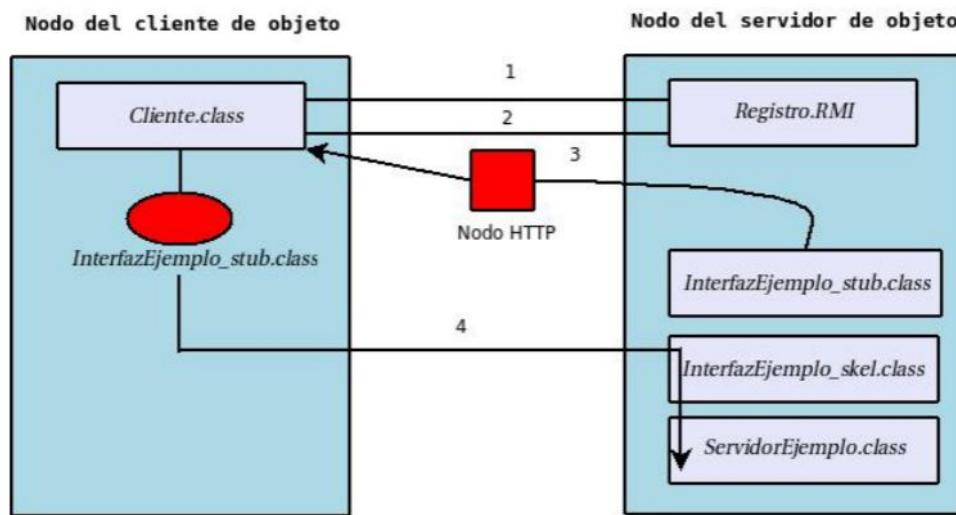
Nota : Los algoritmos de implementación de cliente-server con callbacks, políticas, etc. viene perfecto en el documento del tema 7 RMI en Java : Aspectos Avanzados. Mirar también los códigos de callback que vienen en el campus.



**Figura 1.** Sondeo (polling) frente a callback.

**NO HACE FALTA USAR UN RMI REGISTRY DE PARTE DEL CLIENTE PARA EL USO DE CALLBACKS :**

El contenido de cada tema que va tras las líneas hechas con guiones son información extraída de los documentos del campus, la que va antes de esa línea son contenido de las diapos, el texto en cursiva son respuestas de preguntas de teoría o bastante core de memorizar.



## TEMA 8

### Introducción a Java-RT (real time)

El uso de RTJS requiere modificación en la semántica de la máquina virtual de Java y el sistema de memoria(dividiéndola en más de un tipo de memoria).

- ❖ **Sistemas RT:**
  - Tienen como objetivo dar respuesta a eventos procedentes del mundo real antes de un límite temporal establecido (deadline).
  - Este deadline se cumple independientemente de la carga de trabajo, el nº de procesadores, los hilos y prioridades y los algoritmos de planificación.
- ❖ **Límites de Java Estándar en RT:**
  - Con respecto a la **gestión de memoria**, fragmenta la memoria.
  - La **planificación de Threads** es impredecible, no permite especificar cuándo un hilo ha de ejecutarse.

**El contenido de cada tema que va tras las líneas hechas con guiones son información extraída de los documentos del campus, la que va antes de esa línea son contenido de las diapos, el texto en cursiva son respuestas de preguntas de teoría o bastante core de memorizar.**

- Con respecto a la **sincronización**, la duración de las secciones críticas es **impredecible**. Un hilo no sabe cuántas otras tareas compiten por un recurso ni su prioridad.
  - La gestión asíncrona de eventos no está definida.
  - Una tarea no sabe cuánta memoria física hay ni cuánta **necesita**.
  - En Java estándar tenemos el **garbage collector (gc)**. Como no tenemos acceso a cuando se ejecuta ni podemos nosotros ejecutarlo, no vamos a saber cuanta memoria tenemos disponible en cada momento. Es por eso por lo que no podemos planificar nuestro proceso.
- ❖ **Especificación Java-RT (RTJS-Real Time Java Specification):**
- Java RT elimina el **gc**, pero **baja la productividad del programador**. En JRT es el **programador** quien se encarga de liberar la memoria, al igual q hacemos en C o C++.
  - Engloba el lenguaje Java, no lo modifica.
  - Conserva la compatibilidad hacia atrás, el lenguaje el bytecode. Es decir, es como Java estándar, posee todas sus clases, pero incorpora nuevas funcionalidades.
  - Predictibilidad
  - Incorpora nuevos tipos de hilos, nuevos tipos de memoria
  - Gestión asíncrona de eventos
- Nota: **Garbage Collector (gc)** : Se encarga de administrar de forma automática la memoria, ya que es el encargado de liberar los objetos que ya no están en uso y que no serán usados en el futuro.
- ❖ **Gestión de Memoria API:** JRT provee áreas de memoria donde el gc no afecta ya que estas están fuera del heap.
- El **HEAP** es la memoria asociada al proceso y es la que usamos para el guardado de variables en memoria dinámica. Esta zona de memoria es la zona de memoria que estamos acostumbrados a usar. Las variables que se asignan en memoria dinámica. Aquí si afecta el **gc**.
- En JRT Se utiliza la clase **Abstracta MemoryArea** y sus subclases.
- **HeapMemory**: Aquí afecta el gc.
  - **InmortalMemory**: JRT se crea una nueva zona de memoria: InmortalMemory. A las variables que se declaren aquí no les va a afectar el gc. Se va a quedar esta zona reservada hasta que finalice la aplicación.
  - **ScopedMemory**: Para cuando quiera utilizar una variable durante un tiempo pero quiero que se libere cuando ya no sea necesaria. Aquí en vez de ejecutar el gc cada cierto tiempo, en el momento en que una zona de memoria tenga el contador a 0, inmediatamente se libera la memoria. Ej: Si tengo un vector que apunta a 1000 bytes, en el momento en que deje de apuntar ahí pq sale de su ámbito y el contador se ponga a 0, se libera la memoria.
    - **VTMemory**: Reservar memoria aquí es de complejidad variable. Si me da igual el tiempo que tarde uso VTMemory
    - **LTMemory**: Reservar memoria es de complejidad lineal. Más costosa
- ❖ **Planificación en Java estándar (Limitaciones):**
- No garantiza que los hilos de alta prioridad se ejecuten antes.

**El contenido de cada tema que va tras las líneas hechas con guiones son información extraída de los documentos del campus, la que va antes de esa línea son contenido de las diapos, el texto en cursiva son respuestas de preguntas de teoría o bastante core de memorizar.**

- **Esquema de 10 prioridades** que cambia durante el mapping a prioridades de sistema.
- **El concepto de planificación por prioridad es tan débil que lo hace completamente inadecuado para su uso en entornos RT.**
- Los **hilos de mayor prioridad** eventualmente se ejecutarán **antes** pero **no hay garantías** de que ocurra así.
- **La planificación por prioridades no sirve para nada.** El mapping de hilos entre máquina virtual y SO no es exacto.

❖ **Planificación en Java-RT:**

- **Los niveles de prioridad, a diferencia de Java estándar, no se alteran entre la máquina virtual de java y el SO.**
- **Planificación por prioridades fijas, expansiva** (**Un hilo de mayor prioridad siempre se ejecutará antes que un hilo de menor prioridad incluso si ese hilo de mayor prioridad tiene que sacar de la cpu al hilo de menor prioridad**) y con **28 niveles de prioridad**.
- **Los objetos planificables no cambian su prioridad** salvo si hay **inversión de prioridad** (la inversión de prioridad se produce cuando un hilo de alta prioridad se bloquea esperando un monitor que sea propiedad de un hilo de menor prioridad. ) => **herencia de prioridad**.
- **Expansiva** porque el sistema puede expulsar por diferentes motivos al objeto en ejecución.

**Nota: Planificación expansiva:** Si el planificador es capaz de quitar a un proceso del procesador, la planificación se denomina expansiva.

**Planificación no expansiva:** Cuando un proceso consigue el procesador ya no lo cede hasta que termina.

❖ **Objetos planificables:**

- **En JRT podemos controlar la planificación de los hilos sobre cómo se van a ejecutar.** Aunque esta tiene planificación por prioridad **podemos implementar nuestra propia planificación de hilos**. Para ello usamos **schedulable**.
- **Implementa la interfaz Schedulable y puede ser**
  - **Hilos RT** (clase **RealTimeThread** y **NoHeapRealTimeThread**). Son de la clase **RealTimeThread** que a su vez extiende de **Thread**. Así que **es un tipo de Thread con más funcionalidad**. Solo escribe en la memoria no afectada por el gc.
  - **Gestores de Eventos Asíncronos** (clase **AsyncEventHandler**)
- **Cada objeto debe especificar sus requerimientos temporales** indicando
  - **Requerimientos de lanzamiento (release)**
    - Pueden ser:
      - ◆ **Periódico** (activación por intervalos regulares)
      - ◆ **Aperiódico** (activación aleatoria. Me da igual cuando se ejecute)
      - ◆ **Esporádico** (activación irregular con un tiempo mínimo entre dos activaciones. Ej: que se ejecute como min pasado 1 seg. Que como max haya 1 seg entre 2 activaciones)
    - **Todos tienen un coste y un deadline relativo** donde el **coste** es el **tiempo de CPU requerido para cada activación** y el **deadline** es el **límite de tiempo dentro del cual la activación actual debe haber finalizado**.

**El contenido de cada tema que va tras las líneas hechas con guiones son información extraída de los documentos del campus, la que va antes de esa línea son contenido de las diapos, el texto en cursiva son respuestas de preguntas de teoría o bastante core de memorizar.**

- En los parámetros de lanzamiento puedo pasarle una estimación de lo que va a consumir ese hilo en la cpu. Lo que le paso es una estimación del coste de cpu.

- Requerimiento de memoria (memory)
- Requerimiento de planificación (scheduling)

❖ **Parámetros de planificación:**

- En ocasiones 28 niveles de prioridad pueden ser escasos. Así que podemos tener parámetros de importancia. Dentro de un hilo de la misma prioridad puedo poner que hilo tiene más importancia. Ej: Si tengo dos hilos de prioridad 5 y uno de ellos tiene más importancia, se ejecutará antes el de mayor importancia.
- La importancia solo tiene efecto dentro de la máquina virtual. En el SO no tiene efecto.
- Son utilizados por el planificador para escoger qué objeto planificable ha de ejecutarse.
- RTJS utiliza como criterio de planificación único la prioridad, de acuerdo al gold standard.

❖ **Planificadores:**

- Son los algoritmos responsables de planificar para ejecutar los objetos planificables.
- RTJS soporta planificación expulsiva por prioridades de 28 niveles mediante el PriorityScheduler (subclase de la clase abstracta Scheduler).

*Los Threads Real-Time* son objetos schedulable(planificables) y heredan de la clase Thread.

❖ **Control de la sincronización:**

- Los objetos planificables deben poderse comunicar y sincronizar.
- Todas las técnicas de sincronización basadas en la exclusión mutua suben inversión de prioridades. Así que Java-RT lo soluciona mediante la técnica de herencia de prioridad.
- Herencia de prioridad: Cuando un hilo prioritario está en la CPU y tengo otro hilo de mayor prioridad bloqueado, lo que hago es asignarle al de menor prioridad de manera temporal la máxima prioridad para que no haya inversión de prioridad. Ej: Si tengo un hilo de prioridad 5 en la cpu y otro hilo de prioridad 10 bloqueado, lo que hago es asignar al hilo de prioridad 5 una prioridad 10.

**Nota: Inversión de prioridades:** Se da cuando dos tareas de distinta prioridad comparten un recurso y la tarea de menor prioridad bloquea el recurso antes que la de prioridad mayor, quedando bloqueada esta última tarea en el momento que precise el uso del recurso compartido. Esto hace que queden invertidas de forma efectiva las prioridades relativas entre ambas ya que la tarea que originalmente tenía mayor prioridad queda supeditada a la tarea de menor prioridad. Como consecuencia, la tarea de mayor prioridad puede no cumplir sus requisitos de tiempo establecidos.

Ejemplo de esto: Supongamos que A sea el proceso de mayor prioridad, B una de prioridad intermedia y C la de menor prioridad. C entra a una sección crítica, mientras está en ella es “quitada” —preempted— de la CPU para ejecutar el proceso B que tiene mayor prioridad. En ese momento A se ejecuta e intenta entrar en la misma sección crítica que C, como C está en ella A es bloqueado y el planificador del —el scheduler— sistema operativo vuelve a ejecutar el proceso B.

Es decir, C nunca sale de la sección crítica porque hay un proceso B con mayor prioridad, A que tiene mayor prioridad que los demás tampoco se ejecuta porque está esperando que C salga de la sección crítica... pero ésta no sale porque B tiene preferencia.

El contenido de cada tema que va tras las líneas hechas con guiones son información extraída de los documentos del campus, la que va antes de esa línea son contenido de las diapos, el texto en cursiva son respuestas de preguntas de teoría o bastante core de memorizar.

#### JRTS VS JAVA (En términos de prioridad) Tabla comparativa(Pregunta CORE de Teoría):

	JRTS	JAVA
Número de prioridades	Hasta 28 prioridades	10 prioridades
Garantía de ejecución	JRTS garantiza que los hilos de alta prioridad se ejecuten antes.	Java no garantiza que los hilos de alta prioridad se ejecuten antes.
Tipo de planificación	Expulsiva (capacidad de bloquear un hilo mientras se ejecuta, dando prioridad).	No expulsiva (Imposibilidad de determinar qué proceso será bloqueado, se encarga el SO).
Tipo de prioridad	Fija, ya que los objetos planificables no cambian su prioridad.(salvo si hay inversión de prioridad => herencia de prioridad).	Variable, ya que pueden cambiar su prioridad, según el SO vea oportuno.
Compatibilidad	Sistemas de tiempo real.	Cualquier SO.

---

¿ESTA USTED PREPARAD@ PARA VER 79127039131 CLASES?  
HAY UN POCO DE INFORMACION REPETIDA PERO VA DE LA MANO CON LA EXTRAIDA DEL TEXTO, ÁNIMO QUE ESTO ES LO MÁS DIFÍCIL DE LA ASIGNATURA :)

Los tipos de memoria que se utilizan en RTJS son :

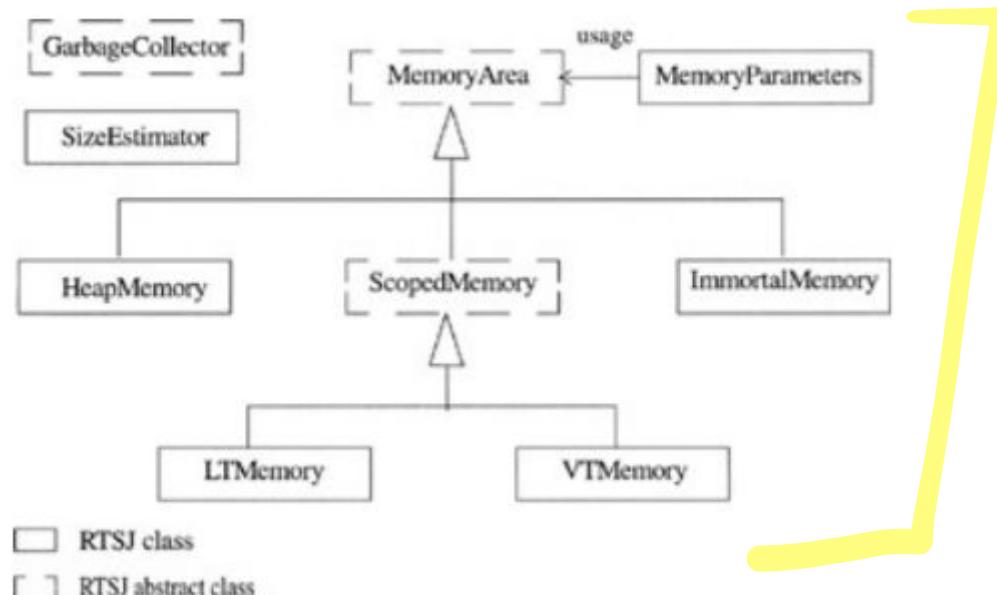
- **Heap memory** : Heap se denota como la cantidad de memoria que emplean la mayoría de lenguajes para las implementaciones run-time (por ejemplo la JVM, la que se encarga de las reservas o liberaciones de memoria) así el programador puede hacer peticiones dinámicas para la reserva(allocation) de fragmentos(chunks, por ejemplo Arrays de los que no se sabe el tamaño en tiempo de compilación). De hecho todos los objetos en Java son reservados en el heap, el garbage collector se ejecuta como una parte de la JVM aunque podría no actuar a pesar de que el heap este lleno.
- **Memory areas** : The RTJS reconoce cuando es necesario permitir gestión de memoria, lo cual no es afectado por la acción del garbage collection(excepto en HeapMemory), las memory áreas son zonas de memoria que están fuera del Heap de Java y NUNCA se verán afectadas por el G.Collector.  
MemoryArea es una clase abstracta de la cual se derivan todas las áreas de memoria de RTJS, el estándar de RTSJ define los siguientes tipos :
  1. **HeapMemory** : Heap memory permite a los objetos estar situados en el standart Java heap. Esta es la zona de memoria afectada por el GC.

El contenido de cada tema que va tras las líneas hechas con guiones son información extraída de los documentos del campus, la que va antes de esa línea son contenido de las diapos, el texto en cursiva son respuestas de preguntas de teoría o bastante core de memorizar.

2. *ImmortalMemory* : *Immortal memory es compartida por todos los hilos del programa Java, el G.C. nunca modifica en este tipo y la memoria se libera solamente cuando el programa termina.*
3. *ScopedMemory* : *Scoped memory es una área de memoria con objetos que tienen una vida claramente definidos. Existe un contador de referencias que indica cuantas entidades Real-Time están usando esta área a la vez, cuando este contador llega a 0 significa que todos los objetos residentes en esta memoria han terminado la ejecución de sus métodos y por lo tanto la memoria está disponible para reutilizarse. The ScopedMemory class es abstracta y tienes varias subclases.*
4. *VTMemory* : Es una subclase de *ScopedMemory* donde las reservas de memoria pueden llevar cantidades variables de tiempo.
5. *LTMemory* : Es una subclase de *ScopedMemory* donde las reservas de memoria se logran en tiempo lineal (es decir, el tiempo de reserva es directamente proporcional al tamaño del objeto).

Los Memory parameters puede utilizarse para la creación de real-time thread y manejadores de eventos asíncronos (asynchronous event handlers), a su vez tamb puede ser cambiados mientras los real-time thread or event handlers estén activos. Estos parámetros especifican :

- La máxima cantidad de reserva de memoria por thread/handler que puede consumir en la memory area por defecto.
- La máxima cantidad de reserva que puede ser utilizada en immortal memory.
- Un límite de ratio de reserva de memoria (bytes per second).



**El contenido de cada tema que va tras las líneas hechas con guiones son información extraída de los documentos del campus, la que va antes de esa línea son contenido de las diapos, el texto en cursiva son respuestas de preguntas de teoría o bastante core de memorizar.**

**Objetos planificados(Schedulable Objects)** : Clases de objetos que implementan la interfaz Schedulable, cada objeto planificado debe indicar de forma específica :

1. Release requirement (cuando tiene que volverse Runnable).
2. Memory requirements : Ej.: La indicación de cuando se va a reservar la memoria en el heap.
3. Scheduling requirements : Ej.: La prioridad a la cual deberían planificarse.

**Parametros que afectan la planificación** : Los Release requirement se especifican a través de herencia de la clase `ReleaseParameters`, la teoría sobre planificación suelen explicar tres tipos de Releases (“liberaciones” o lanzamiento de tareas) :

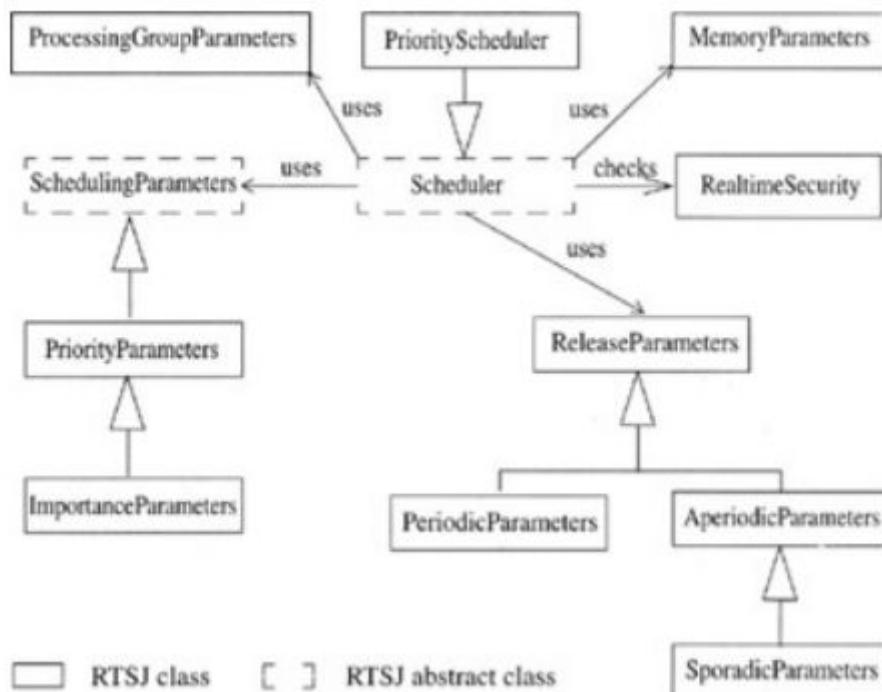
- **Periódicas( PeriodicParameters )** : Released(lanzar tarea) en un “tiempo” regular. Una vez pasado este tiempo tras el lanzamiento, se volverá a lanzar, y así sucesivamente.
- **Aperiódicas( AperiodicParameters )** : Released en un momento aleatorio. Se volverán a lanzar cada tiempo aleatorio.
- **Esporádicas( AperiodicParameters )** : Released irregularmente pero con un tiempo mínimo entre cada release(lanzamiento).

Todos las clases `ReleaseParameters` encapsulan un Coste y un valor `Deadline`(tiempo relativo), The cost is the maximum amount of CPU time (execution time) needed to execute the associated schedulable object every time it is released. La deadline es el tiempo para el cual el objeto debe haber terminado su ejecución, es especificado relativo al tiempo de lanzamiento “Realease” del objeto. Además `PeriodicParameters` tamb incluyen el `start time` del primer lanzamiento y del primer intervalo entre lanzamientos(`period`). Y `SporadicParameters` incluyen el tiempo mínimo entre lanzamientos “Releases”.

The abstract class `SchedulingParameters` provides the root class from which a range of possible scheduling criteria can be expressed. `ImportanceParameters` allow an additional numerical scheduling metric to be assigned; it is a subclass of the `PriorityParameters` class.

**Planificadores(Schedulers)** : Son los responsables de la planificación de los objetos planificables(NO confundir con objetos planificados). El RTSJ no indica cuantos planificadores puede existir en una misma real-time JVM pero comúnmente solo uno estará presente. Aunque RTSJ soporta la planificación basada en prioridades indicada de forma EXPLÍCITA con `PriorityScheduler` por lo que `Scheduler` es una clase abstracta con `PriorityScheduler` como una clase abstracta definida.

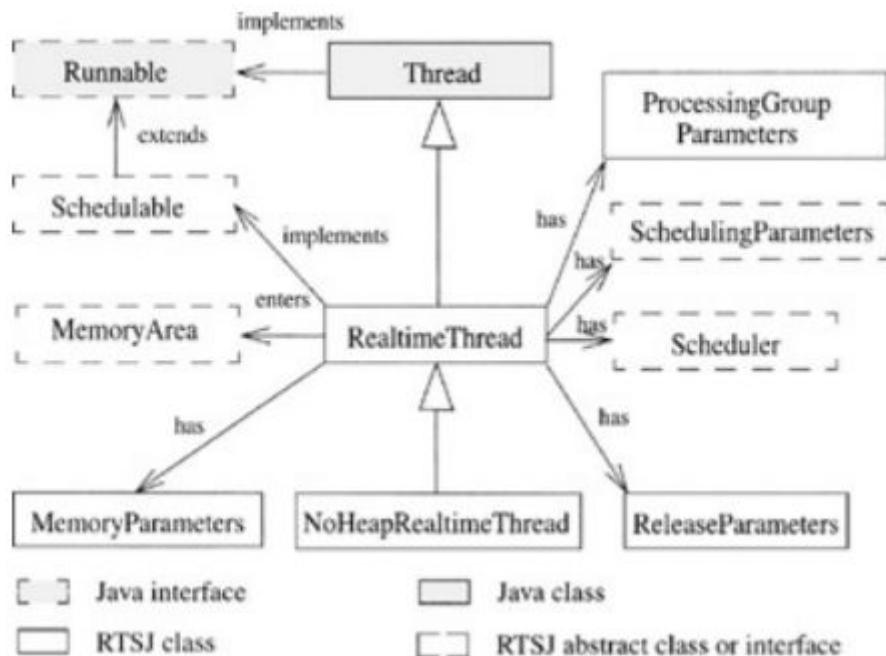
El contenido de cada tema que va tras las líneas hechas con guiones son información extraída de los documentos del campus, la que va antes de esa línea son contenido de las diapos, el texto en cursiva son respuestas de preguntas de teoría o bastante core de memorizar.



Veámoslo de esa forma, un planificador utilizará :

- Parámetros de lanzamiento.
- Parámetros de planificación.
- Un grupo de parámetros de Procesamiento.
- Parámetros de memoria
- Comprobará la seguridad en tiempo real.
- Modelarán planificadores de prioridad.

#### Real-Time Threads (Classes):



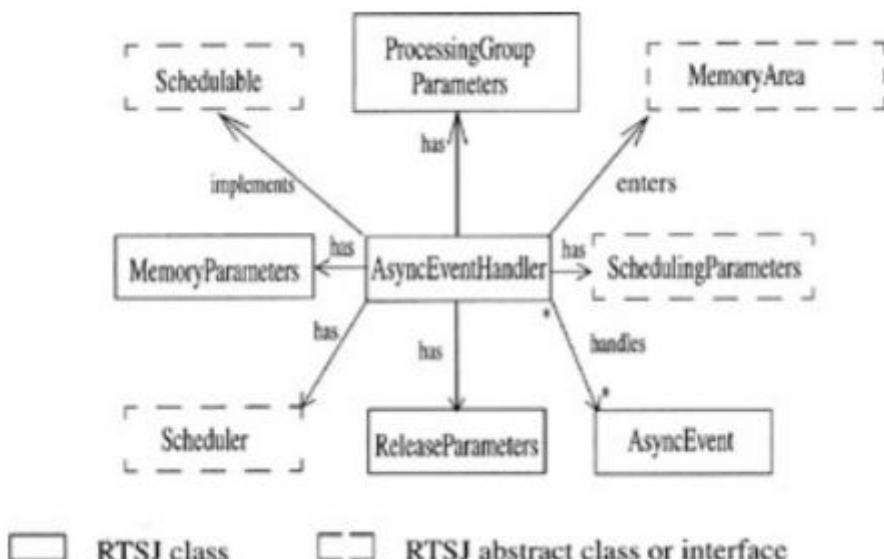
**El contenido de cada tema que va tras las líneas hechas con guiones son información extraída de los documentos del campus, la que va antes de esa línea son contenido de las diapos, el texto en cursiva son respuestas de preguntas de teoría o bastante core de memorizar.**

A periodic real-time thread is a real-time thread that has periodic release parameters. Similarly, an aperiodic (or sporadic) real-time thread is one that has aperiodic (or sporadic) release parameters. A `NoHeapRealtimeThread` is one that guarantees not to create or reference any objects on the heap.

Veamoslo así, un hilo de tiempo real tiene :

- ❖ Un planificador.
- ❖ Unos parámetros de planificación.
- ❖ Parámetros de lanzamiento.
- ❖ Parámetros de memoria
- ❖ Un grupo de parámetros de Procesamiento.
- ❖ Son de carácter planificables.
- ❖ Modelan los hilos `NoHeapRtT`
- ❖ Son modelados por la clase `Thread`

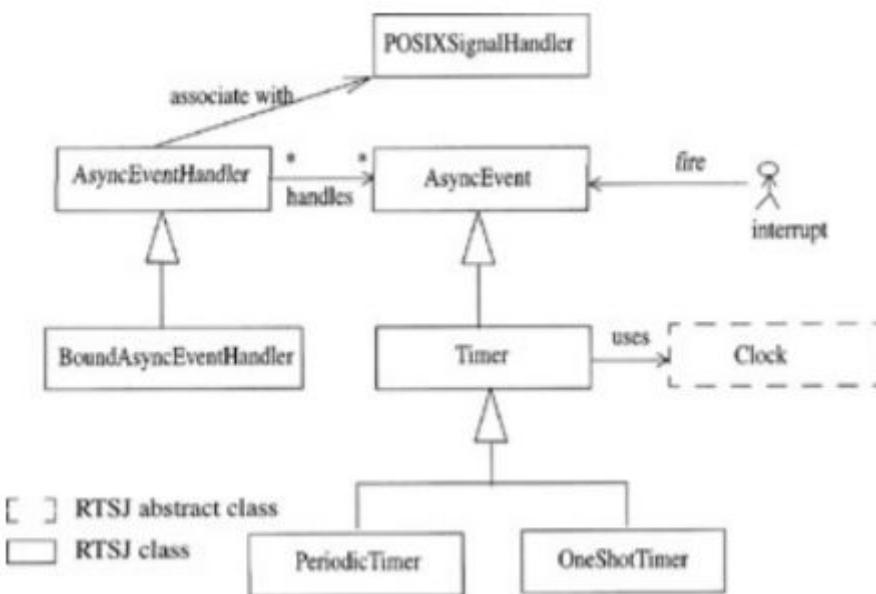
#### **Asynchronous Event Handling and Timers :**



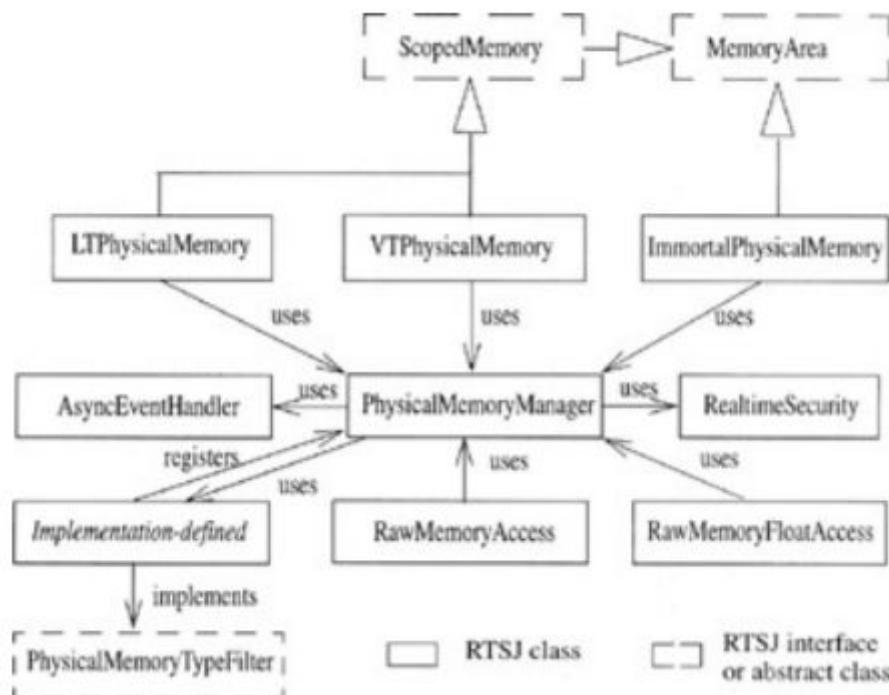
Cada `AsyncEvent` puede tener uno o varios manejadores, y un mismo manejador puede estar asociado a un evento o más. **RUNNABLE MODELA LA INTERFAZ SCHEDULABLE Y TMB A ASYNCEVENTHANDLER.**

La relación entre Timer y los eventos sería :

El contenido de cada tema que va tras las líneas hechas con guiones son información extraída de los documentos del campus, la que va antes de esa línea son contenido de las diapos, el texto en cursiva son respuestas de preguntas de teoría o bastante core de memorizar.



El texto entre la imagen de arriba y la de abajo son conceptos avanzados de RTSJ que un@ se mira si va sobrad@ de tiempo, la siguiente imagen es un poco locura la vd :



**Sincronización** : Mutual exclusion without blocking – supported by the wait-free communication classes.

## COSAS DE JAVA

El callable es como el runnable pero el run devuelve algo. En vez del método run tenemos el método call. Esto devolverá un **java.util.concurrent.Future**, que permite comprobar el estado de un Callable.

**El contenido de cada tema que va tras las líneas hechas con guiones son información extraída de los documentos del campus, la que va antes de esa línea son contenido de las diapos, el texto en cursiva son respuestas de preguntas de teoría o bastante core de memorizar.**

The get method is a synchronous method. Until the callable finishes its task and returns a value, it will wait for a callable.

*Usa bloqueos implícitos al acceder al resultado.*

Ejemplo:

```
package com.arquitecturajava;
import java.util.concurrent.Callable;
public class MiCallable implements Callable<Integer> {
    @Override
    public Integer call() throws Exception {
        int total = 0;
        for(int i=0;i<5;i++) {
            total+=i;
            try {
                Thread.sleep(300);
            } catch (InterruptedException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }
        }
        System.out.println(Thread.currentThread().getName());
        return total;
    }
}

package com.arquitecturajava;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;

public class PrincipalCallable {
    public static void main(String[] args) {

        try {
            ExecutorService servicio= Executors.newFixedThreadPool(1);
            Future<Integer> resultado= servicio.submit(new MiCallable());
            if(resultado.isDone()) {
                System.out.println(resultado.get());
            }
        } catch (InterruptedException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        } catch (ExecutionException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }
}
```

- **Submit:** produce un objeto Future devuelto por el objeto Callable.

- **isDone():** del objeto Future para ver si se ha completado.

- **get()** para extraer el resultado. Se bloqueará hasta que el resultado esté listo.

**Ecuación de subramanian:**

**El contenido de cada tema que va tras las líneas hechas con guiones son información extraída de los documentos del campus, la que va antes de esa línea son contenido de las diapos, el texto en cursiva son respuestas de preguntas de teoría o bastante core de memorizar.**

$$N_t = \frac{N_{nld}}{1 - C_b}$$

Sirve para determinar el número de hilos necesarios en función del número de núcleos lógicos disponibles y del coeficiente de bloqueo del problema.

**Runtime.getRuntime().availableProcessors();** -> nº de núcleos lógicos

#### Números aleatorios:

La llamada a *Math.random()* devuelve un número aleatorio entre 0.0 y 1.0, excluido este último valor. Devuelve un double.

*Math.random() \* n* donde n es el número máximo. -> Ejemplo: *Math.random() \* 6* para que salgan números entre 0 y 5.

Si queremos poner entre 1 y 6 incluido poner un +1 -> *Math.random() \* 6 + 1*

Si queremos entre dos números por ejemplo 5 y 10 ambos incluidos, sería  
*Math.random() \* (maximo-minimo+1) + mínimo*

Se recomienda obtener números aleatorios con objetos de la clase Random:

Ex: Random r = new Random();  
r.nextInt() / r.nextFloat() / r.nextInt()...

#### PARALELISMO GRANO LISO Y GRANO GRUESO

**De grano fino:** es cuando el código se divide en una gran cantidad de piezas pequeñas, es a un nivel de sentencia donde un ciclo se divide en varios subciclos que se ejecutarán en paralelo se le conoce además como Paralelismo de Datos.

**De grano grueso:** es a nivel de subrutinas o segmentos de código, donde las piezas son pocas y de cómputo más intensivo que las de grano fino se le conoce como Paralelismo de Tareas

#### SPEED UP entre Java y C++

T. C++/ T. JAVA < 1 C++ Tarda Menos

Para calcular el Tiempo de C++:

```
#include <iostream>
#include <ctime>

unsigned t0, t1;

t0=clock()
// Code to execute
```

**El contenido de cada tema que va tras las líneas hechas con guiones son información extraída de los documentos del campus, la que va antes de esa línea son contenido de las diapos, el texto en cursiva son respuestas de preguntas de teoría o bastante core de memorizar.**

**t1 = clock();**

```
double time = (double(t1-t0)/CLOCKS_PER_SEC);
cout << "Execution Time: " << time << endl;
```

**Para calcular el Tiempo de Java:**

```
long time_start, time_end;
time_start = System.currentTimeMillis();
ReallyHeavyTask(); // Llamamos a la tarea
time_end = System.currentTimeMillis();
System.out.println("the task has taken "+ ( time_end - time_start ) +" milliseconds");
```

## OTRAS COSAS Q NO SE DONDE PONERLAS

Un programa **determinista** es aquella en la que los mismos datos de entrada generan siempre la misma salida. **no determinista** es aquella que dada una entrada produce diferentes salidas.

**Pregunta exámen (no me acuerdo bien del enunciado. Sacado de la revisión) :** Cubo de datos. Tenemos una matriz tridimensional (cubo), y te dicen que existe interdependencia entre los hilos porque los hilos como no puede haber otro cubo, no tiene memoria para hacer otro cubo, ni otra matriz ni nada, y tienes que modificar el mismo cubo. Si una hebra quiere modificar una celda, y otra hebra quiere modificar otra celda, habría un conflicto ya que están intentando modificar a la vez el mismo cubo por lo que alguna de las dos hebras tendría que bloquearse. Por lo cual si al menos una hebra se va a bloquear lo ideal sería tener más hebras disponibles para que mientras esa hebra esté bloqueada otra hebra pudiese ocupar su sitio en un núcleo de la cpu. Así que la respuesta es: Crear más hebras que núcleos tenga la arquitectura.

\*Nota: En el momento en el que el problema te de a entender que pueda haber hebras bloqueadas, necesitas más hebras que núcleos. Es como si el  $c_B$  fuese  $> 0$ . Si aplicamos la fórmula ya nos sale más hilos que núcleos.

**Pregunta exámen: El método start de la clase thread puede ser invocado sobre objetos que expresan? concurrencia implementando la interfaz runnable.**

Respuesta: No, porque directamente sobre objetos que implementan la interfaz runnable no. Ten en cuenta que los objetos que implementan esta interfaz tienen que estar encapsulados dentro de un objeto de tipo thread. Cuando creamos un hilo de esta clase: Clase h1 = new Clase (a1,a2,a3); luego tiene que estar encapsulado en un objeto de tipo thread -> Thread hilo = new Thread (h1) y mediante ese objeto de tipo thread ya le haces hilo.start().

**Pregunta exámen (desarrollo sept 2018):Cerrojos vs Monitores**

Respuesta: Con cerrojos tienes un nivel de abstracción mucho menor, puedes ir más al detalle y puedes bloquear a los procesos de una forma más fina. Con un monitor bloqueas al recurso completo.. Por ejemplo, tenemos un monitor para productores-escritores. Tienes un array tanto para leer como para escribir. Con los cerrojos, puedes bloquear a los que están intentando extraer o a los que están intentando producir. Así que sería un poco más eficiente con los cerrojos. Cuanto más bajo sea el nivel de abstracción, más eficiente es la primitiva.

**El contenido de cada tema que va tras las líneas hechas con guiones son información extraída de los documentos del campus, la que va antes de esa línea son contenido de las diapos, el texto en cursiva son respuestas de preguntas de teoría o bastante core de memorizar.**

**Pregunta exámen (desarrollo sept 2018): Considera la API de alto nivel para concurrencia. ¿En qué situaciones es más adecuado el uso de la clase Semáforo en lugar de los cerrojos (ReentrantLock)?**

Respuesta: Con los cerrojos no puedes hacer sincronización. Los cerrojos solo sirven para hacer exclusión mutua. O permites que entren o no. Pero hay problemas que requieren una sincronización más avanzada. Con los semáforos, si tienes un semáforo inicializado a 1, consigues el mismo efecto que con un cerrojo pero además con los semáforos puedes hacer otro tipo de sincronización.

Puedes permitir por ejemplo que haya dos procesos dentro o puedes hacer que varios procesos se esperen a que termine otro. Es decir con los semáforos consigues una sincronización más genérica mientras que con los cerrojos solo puedes hacer exclusión mutua. (La única sincronización que tiene es la de q solo 1 proceso puede estar en la sección crítica). Los cerrojos solo te permiten un tipo de sincronización que es el de exclusión mutua.

¿Cuál usar? Depende del problema. En aquellos problemas en los que requieran una sincronización más específica usar semáforos.

Por ejemplo, para un problema como el de los filósofos que requieren una sincronización más específica es mejor usar los semáforos

**Pregunta exámen (desarrollo sept 2018): Era algo de hacer una gráfica comparando el paralelismo de grano grueso, de grano fino y el secuencial..**

La idea es que cuando tu tienes que solucionar un problema con hebras, si utilizamos paralelismo de grano grueso y utilizamos pocas tareas va a ir más rápido que si por ejemplo usamos una programación de grano fino porque con grano fino tendríamos tantas tareas como celdas. Al final la sobrecarga de gestionar todas esas hebras haría que fuese más lento que el de grano grueso.

Luego tenemos el secuencial que estaría entre el grano fino y el grano grueso (en el medio). No tiene tanta sobrecarga como el de grano fino por tener tantas hebras pero va más lento que el concurrente.

**Pregunta exámen (desarrollo sept 2018): ¿Por qué hace falta una clase noHeapRealTime? (lo q es el enunciado era de otra forma pero se resume en esa pregunta)**

NoHeapRealTimeThread lo que hace es que las hebras que utilizan esa clase de base no guardan datos en la memoria estándar de la máquina virtual de java, es decir, no le afecta el recolector de basura de java, entonces si en un sistema de tiempo real necesitas tener certeza de la memoria que vas a tener disponible en cada instante necesitas que no esté el recolector de basura que esté por ahí haciendo lo que le de la gana. Lo que necesitamos es reservar memoria y liberar memoria cuando queramos. Así que para eso necesitamos meterlo fuera del Heap en java. Hay una zona específica que no es afectada por el recolector de basura.

(Mirar también el diagrama)

**Pregunta exámen (desarrollo sept 2018): La última pregunta era sobre las instrucciones de la GPU. (Mirar documento del seminario que está explicado)**

**Conceptos del examen:**

Multiprocesamiento simétrico (SMP): Es un tipo de arquitectura de computadoras en la que dos o más unidades de procesamiento comparten una única memoria central compitiendo en igualdad de

**El contenido de cada tema que va tras las líneas hechas con guiones son información extraída de los documentos del campus, la que va antes de esa línea son contenido de las diapos, el texto en cursiva son respuestas de preguntas de teoría o bastante core de memorizar.**

condiciones por dicho acceso, de ahí la denominación "simétrico". Salguero dijo: "dividir el trabajo. Si tienes 2 hilos lo divides en 2, si tienes 4 hilos, lo divides en 4".

Clase segura: Es básicamente un monitor. Por ejemplo: Si tenemos una red local con ordenadores conectados un plotter de impresión. Lo que podemos hacer es crearnos un monitor al que tu le pidas que tu le pidas una impresora y ya es esa clase la que se encarga de hacer la gestión de la concurrencia de los plotter que tiene disponible.

Una pregunta era sobre algo de aplicando las condiciones de seguridad de Bernstein.

**Condiciones de Bernstein:** Para determinar si dos conjuntos de instrucciones se pueden ejecutar concurrentemente se definen:

$L(S_k) = \{a_1, a_2, \dots, a_n\}$  conjunto de lectura del conjunto de instrucciones  $S_k$ , formado por todas las variables cuyos valores son leídos (referenciados) durante la ejecución de las instrucciones en  $S_k$ .

$E(S_k) = \{b_1, b_2, \dots, b_n\}$  conjunto de escritura del conjunto de instrucciones  $S_k$ , formado por todas las variables cuyos valores son actualizados (se escriben) durante la ejecución de las instrucciones en  $S_k$ .

Para que dos conjuntos de instrucciones  $S_i$  y  $S_j$ ,  $i \neq j$ ,  $i < j$  se puedan ejecutar concurrentemente se tiene que cumplir que:

1.  $L(S_i) \cap E(S_j) = \text{vacío}$
2.  $E(S_i) \cap L(S_j) = \text{vacío}$
3.  $E(S_i) \cap E(S_j) = \text{vacío}$

Ejemplo:

**El contenido de cada tema que va tras las líneas hechas con guiones son información extraída de los documentos del campus, la que va antes de esa línea son contenido de las diapos, el texto en cursiva son respuestas de preguntas de teoría o bastante core de memorizar.**

- Sean: Se calculan los conjuntos de lectura y escritura

$$S_1 \rightarrow a := x + y; \quad \begin{array}{l} L(S_1) = \{x, y\} \\ E(S_1) = \{a\} \end{array}$$

$$S_2 \rightarrow b := z - 1; \quad \begin{array}{l} L(S_2) = \{z\} \\ E(S_2) = \{b\} \end{array}$$

$$S_3 \rightarrow c := a - b; \quad \begin{array}{l} L(S_3) = \{a, b\} \\ E(S_3) = \{c\} \end{array}$$

$$S_4 \rightarrow w := c + 1; \quad \begin{array}{l} L(S_4) = \{c\} \\ E(S_4) = \{w\} \end{array}$$

- Sean: **Aplicando las condiciones de Bernstein**

$$L(S_1) = \{x, y\}$$

$$E(S_1) = \{a\}$$

$$L(S_2) = \{z\}$$

$$E(S_2) = \{b\}$$

$$L(S_3) = \{a, b\}$$

$$E(S_3) = \{c\}$$

$$L(S_4) = \{c\}$$

$$E(S_4) = \{w\}$$

$$\text{Entre } S_1 \text{ y } S_2$$

$$1. \quad L(S_1) \cap E(S_2) = \emptyset$$

$$2. \quad E(S_1) \cap L(S_2) = \emptyset$$

$$3. \quad E(S_1) \cap E(S_2) = \emptyset$$

$$\text{Entre } S_1 \text{ y } S_3$$

$$1. \quad L(S_1) \cap E(S_3) = \emptyset$$

$$2. \quad E(S_1) \cap L(S_3) = \{a\} \neq \emptyset$$

$$3. \quad E(S_1) \cap E(S_3) = \emptyset$$

$$\text{Entre } S_1 \text{ y } S_4$$

$$1. \quad L(S_1) \cap E(S_4) = \emptyset$$

$$2. \quad E(S_1) \cap L(S_4) = \emptyset$$

$$3. \quad E(S_1) \cap E(S_4) = \emptyset$$

---

## EJERCICIOS DE LAS PRÁCTICAS QUE HAN CAÍDO EN EXÁMENES ANTERIORES :

Práctica 3 : Ejercicio 4.

Práctica 4 : Ejercicio 1 + particionado del problema con uso de ejecutores.

Práctica 6 : Ejercicio 1(uso de la Ec. de Subramanian,tanto en teoría como en práctica).Ejercicio2,Ejercicio 3(en teoría),Ejercicio 4.

Práctica 7 : Ejercicio 3(Tiene el ejercicio Montecarlo y además de Future & Callable).

Práctica 10 : Ejercicio 1,Ejercicio 2(no exactamente ese enunciado pero si usar ReentrantLock y variables de condicion).

Práctica 11 : Ejericico 1.De todas formas RMI cae **BASTANTE**.

---

**El contenido de cada tema que va tras las líneas hechas con guiones son información extraída de los documentos del campus, la que va antes de esa línea son contenido de las diapos, el texto en cursiva son respuestas de preguntas de teoría o bastante core de memorizar.**

PD : El pack PCTR son este documento,el doc de seminarios,el pack de exámenes del campus comentados y el zip de códigos(conceptos,implementaciones en varios lenguajes y exámenes de prácticas).