I. Conceptos teóricos

Empezar con un capítulo titulado “Conceptos teóricos” quizá no es el mejor reclamo en un libro sobre programación, pero me temo que adquirir una base teórica es imprescindible antes de ponerse a escribir líneas de código. Conocer los conceptos que se explican en este capítulo nos permitirá asentar los fundamentos básicos que nos ayudarán a entender mejor el resto de los capítulos y nos permitirá también expresarnos con propiedad.

En las siguientes páginas, hablaremos sobre qué son los programas y procesos. Qué diferencias hay entre los términos procesador, microprocesador y núcleo. Cómo podemos clasificar los sistemas en función de su capacidad de trabajar o no con múltiples procesos de forma simultánea. Veremos también los fundamentos sobre cómo se gestionan los procesos en los sistemas multitarea. Finalmente, trataremos aspectos relacionados con la programación concurrente, el uso de hilos, la programación paralela y la distribuida. Que no cunda el pánico, será rápido y espero que indoloro.

Programas y procesos

Empezaremos con dos términos sencillos que, aunque utilizamos de forma indistinta, siendo estrictos son muy diferentes: Programa y proceso.

Un **programa** es un conjunto de instrucciones almacenadas en un fichero. Sin embargo, un **proceso** es una instancia de un programa en ejecución, es decir, el proceso se crea cuando ejecutamos un programa. Mientras que las instrucciones y los datos de los programas se almacenan en memoria secundaria (disco duro), las instrucciones y datos de un proceso se encuentran en la memoria principal (memoria RAM) o en los registros internos de la CPU.

Que ejecutemos un programa no implica que se cree un único proceso ya que se pueden lanzar múltiples. Por ejemplo, el navegador *Chrome* crea un proceso por cada pestaña que abrimos.

Procesadores, microprocesadores y núcleos

Procesador, microprocesador y núcleo son tres términos que muchas veces también se utilizan de forma indistinta. Veamos en qué se diferencian.

Como ya sabrás, el componente físico del ordenador que se encarga de ejecutar las instrucciones de los procesos es el **procesador o CPU** (*Central Processing Unit* o Unidad Central de Proceso). Un procesador se compone de varios elementos: unidad de control, unidad aritmeticológica, registros, etc. Al ser componentes tan complejos originalmente se construían mediante diferentes chips de forma que cada uno de ellos realizaba una tarea diferente.

En 1971 se desarrolló el primer **microprocesador**, que no era más que la implementación de un procesador o CPU en un único chip. Sin embargo, hoy en día utilizamos los dos términos, procesador y microprocesador, de forma indistinta.

Si analizamos las características de cualquier microprocesador actual, veremos que incluyen varios procesadores o CPU que reciben el nombre de **núcleos**. Los diferentes núcleos de un procesador permiten que se puedan ejecutar múltiples procesos de forma simultánea.

Sistemas monoproceso y multiproceso

En función de la capacidad que tiene un sistema para trabajar con diferentes procesos, hablamos de sistemas monoproceso o multiproceso.

Los **sistemas monoproceso** son aquellos en los que únicamente se puede ejecutar un proceso a la vez. Es decir, estos sistemas no permiten que en un mismo instante de tiempo se estén ejecutando dos procesos diferentes.

Por ejemplo, los microprocesadores *Intel Pentium* disponían de un única CPU, por lo que eran microprocesadores monoproceso. El sistema operativo *Windows 95* también era un sistema monoproceso porque no soportaba la ejecución de varios procesos de forma simultánea.

Figura 1: A la izquierda procesador Intel Pentium 4. A la derecha logotipo de Windows 95.

Los **sistemas multiproceso** son aquellos que permiten la ejecución de múltiples procesos de forma simultánea. Es decir, en un instante determinado de tiempo pueden existir múltiples procesos en ejecución. Los procesadores que disponen de múltiples núcleos son sistemas multiproceso. Como hemos mencionado anteriormente, un núcleo es una CPU, por lo que un microprocesador de 4 núcleos tiene integradas 4 CPU en su interior, permitiendo la ejecución de hasta 4 procesos de forma simultánea.

Sistemas monotarea y multitarea

Veamos una nueva forma de clasificar los sistemas. Según su capacidad para alternar la ejecución de procesos podemos hablar de sistemas monotarea y multitarea.

Los primeros ordenadores eran **sistemas monotarea**, donde el procesador no podía intercalar la ejecución de diferentes procesos. Hasta que no finalizaba la ejecución de un proceso no podía iniciarse la ejecución

del siguiente. De hecho, en la memoria de estos sistemas se alojaba un único proceso, el que se encontraba en ejecución en ese momento. En estos sistemas se infrautilizaba el procesador, ya que durante las ráfagas de E/S de los procesos el procesador quedaba ocioso, es decir, a la espera de más instrucciones para su ejecución.

¿Qué son las ráfagas de E/S?

Los periodos de tiempo en los que las instrucciones de un proceso se ejecutan en la CPU reciben el nombre de ráfagas de CPU. Sin embargo, hay períodos en los que el proceso no puede continuar su ejecución. Esto ocurre cuando realiza una petición de entrada / salida como puede ser la lectura o escritura de un archivo o la comunicación con un periférico. Puesto que los periféricos, discos duros, memoria principal y en general cualquier componente del ordenador es mucho más lento que el procesador, éste se ve obligado a esperar, quedando ocioso. Estos periodos reciben el nombre de ráfagas de entrada/salida o E/S.

En la siguiente figura se muestra la ejecución de un proceso en un sistema monotarea. Como se puede ver, cuando se produce una ráfaga de E/S el procesador queda ocioso.

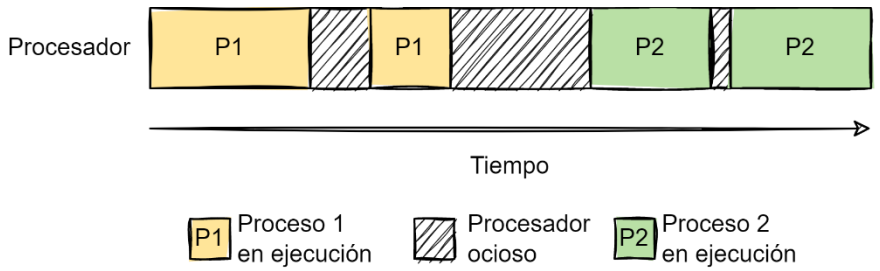


Figura 2: Sistema monoprocesador monotarea.

Para solucionar este problema se desarrolló en los años 60 una técnica denominada **multiprogramación**. Esta técnica permite alojar en la memoria principal múltiples procesos y ejecutarlos de forma intercalada, dando lugar a los sistemas multitarea.

En los **sistemas multitarea** varios procesos pueden intercalar su ejecución: Como se puede ver, en la siguiente figura, cuando un proceso se encuentra en una ráfaga de E/S, el sistema operativo pasa a ejecutar otro proceso. La ejecución de los procesos se intercala a tal velocidad, que da la impresión de que se ejecutan en paralelo (simultáneamente), aunque en realidad lo hacen uno detrás de otro. Gracias a esta técnica se aprovecha mejor la CPU, ya que se reduce el tiempo que permanece ociosa.

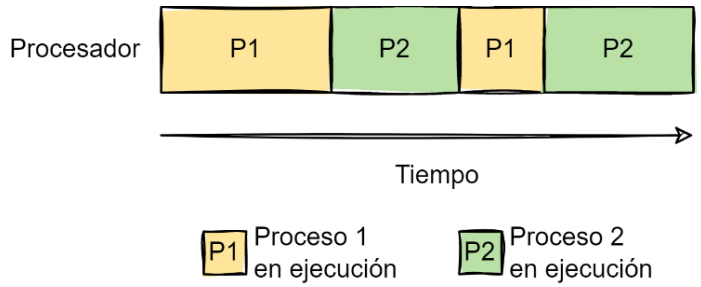


Figura 3: Sistema monoprocesador multitarea.

Hay que tener en cuenta que un sistema monoproceso puede ser monotarea o multitarea. Si es monotarea utilizará un único procesador y no podrá ejecutar el siguiente proceso hasta que no finalice el anterior. Si es multitarea utilizará un único procesador, que será capaz de intercalar en su ejecución diferentes procesos, aunque nunca ejecutaría dos procesos de forma simultánea.

De la misma forma, un sistema multiproceso puede ser monotarea o multitarea. Si es monotarea cada procesador podrá ejecutar un proceso diferente (cada proceso se ejecutará simultáneamente), pero hasta que no se finalice la ejecución de cada proceso, no se podrá ejecutar el siguiente. Si es multitarea cada procesador podrá intercalar en su ejecución diferentes procesos.

Para entender mejor todos estos conceptos utilizaremos un paralelismo con la cocina de un restaurante. La elaboración de un plato del menú requiere que el cocinero (procesador) realice una serie de pasos (tareas o procesos). Por ejemplo, para cocinar un plato de pasta a la boloñesa, hay que sofreír la carne, cortar la cebolla, hervir la pasta, etc.

Un **sistema monoprocesador y monotarea** equivaldría a tener un único cocinero que es incapaz de realizar varias tareas a la vez. Primero pone la pasta a hervir y se queda mirando a la espera de que termine la cocción. Cuando finaliza, pasa a realizar la siguiente tarea: sofreír la carne. De nuevo, hasta que la carne no está sofrita, no pasa al siguiente paso. Es evidente que estos sistemas son poco eficientes. El sistema mostrado en la Figura 2, aparte de ser monotarea, es monoprocesador, ya que dispone de un único procesador. Un ejemplo de sistema operativo monoprocesador y monotarea es el sistema operativo *MSDOS*.

Un **sistema monoprocesador y multitarea** equivaldría a tener un único cocinero que es capaz de realizar varias tareas a la vez. Primero pone la pasta a hervir, y mientras se calienta el agua, pone la carne a sofreír y corta la cebolla. El sistema mostrado en la Figura 3, aparte de ser multitarea, es monoprocesador, ya que dispone de un único procesador. Un ejemplo de sistema operativo monoprocesador y multitarea es *Windows 3.11* y *Windows 95*. En estos sistemas operativos el usuario ya podía realizar varias tareas como escuchar música mientras escribía un documento de texto.

Un **sistema multiprocesador y monotarea** equivaldría a tener varios cocineros que son incapaces de realizar varias tareas a la vez. Así, un cocinero pone la pasta a hervir y espera a que se cueza. Mientras tanto, el otro cocinero sofríe la carne y también queda a la espera. Hasta que no terminen sus pasos, ninguno de los dos cortará la cebolla. En la Figura 4 se puede ver la representación de varios procesos ejecutándose en un sistema multiprocesador y monotarea.

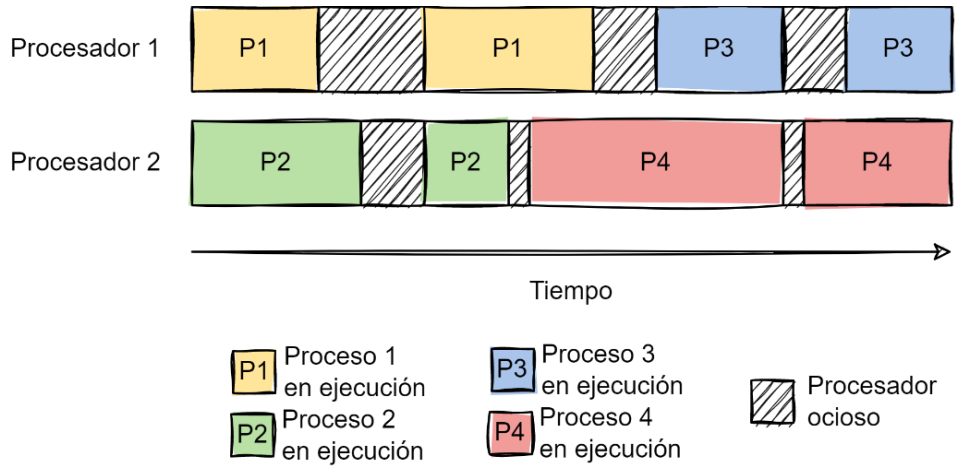


Figura 4: Sistema multiprocesador y monotarea.

Un sistema multiprocesador y multitarea equivaldría a tener varios cocineros que son capaces de realizar varias tareas a la vez. Así, un cocinero es capaz de hervir la pasta y sofreír la carne, mientras el otro corta la cebolla y realiza cualquier otra tarea. En la Figura 5 se representa la ejecución de procesos en un sistema de este tipo.

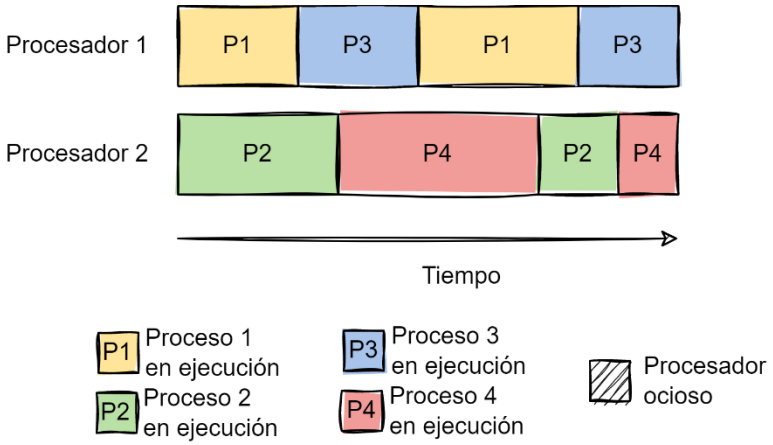


Figura 5: Sistema multiprocesador y multitarea.

Cualquier ordenador de sobremesa, portátil o teléfono móvil actual es multiproceso y multitarea. Podríamos pensar que, hoy en día, todos los sistemas son multiproceso y multitarea, pero hay que tener en cuenta que los procesadores están en todas partes ¡Hasta en las tostadoras! Y no siempre son sistemas que necesitan ejecutar una gran cantidad de procesos.

Gestión de procesos en sistemas multitarea

Ahora bien, ¿Cómo gestiona el sistema operativo la ejecución de los procesos en los sistemas multitarea? Sin duda es una tarea compleja, así que daremos una visión general. Las principales tareas que lleva a cabo el sistema operativo para gestionar los procesos son:

- **Creación de procesos**: Cuando se ejecuta un programa carga las instrucciones y datos en memoria RAM. Para cada proceso crea un registro denominado Bloque de Control de Proceso (BCP), donde se almacena toda la información relativa al proceso.
- **Eliminación de procesos**: Cuando la ejecución de un proceso finaliza, el sistema operativo libera la memoria y los recursos utilizados por el proceso.
- **Planificación de la ejecución de los procesos**: Decide qué proceso se ejecuta en cada momento. Esto ocurre, por ejemplo, cuando un proceso se encuentra en una ráfaga de E/S. Para que el procesador no quede ocioso, el sistema operativo desaloja el proceso y elige uno de los procesos en espera para que se ejecute. La elección de cuál será el siguiente proceso que se ejecutará la realiza el **planificador de procesos**.
- **Cambios de contexto**: Una vez elegido cuál será el siguiente proceso que se ejecutará, hay que desalojar el que se está ejecutando actualmente, guardar sus datos en el BCP y cargar el BCP del nuevo proceso.
- **Asignación de recursos:** Los procesos requieren el uso de memoria principal, acceso a archivos y el uso de periféricos. El sistema operativo se encarga de reservar estos recursos y asignarlos al proceso.

Bloque de Control de Proceso

Como hemos mencionado en el apartado anterior, el sistema operativo utiliza un registro denominado Bloque de Control de Proceso (BCP) para almacenar toda la información relativa del proceso. En este registro se almacena, entre otros datos:

- **Identificador del proceso**: Número que identifica de forma unívoca al proceso.
- **Estado del proceso**: Indica el estado en el que se encuentra el proceso. Veremos más adelante cuáles son estos estados.
- **Contador de programa**: Indica la dirección de memoria de la siguiente instrucción a ejecutar.
- **Registros de la CPU**: Valor de los registros de la CPU.
- **Información de planificación de la CPU.**
- **Información de gestión de memoria.**
- **Información sobre la entrada / salida**: Dispositivos asignados, archivos abiertos, etc.

Estados de un proceso

Ya hemos visto que los procesos se crean, se ejecutan, pueden permanecer a la espera o ser finalizados, es decir, pasan por diferentes etapas en lo que se llama su ciclo de vida. Estas etapas son:

- **Nuevo**: El proceso acaba de ser creado.
- **Preparado**: El proceso está listo para ser ejecutado, permanece a la espera de que el sistema operativo le asigne CPU.

- **Ejecución**: Sus instrucciones se encuentran en ejecución en el procesador.
- **Bloqueado**: El programa no ha terminado de ejecutarse, pero permanece a la espera de que ocurra un evento externo como puede ser la finalización de una operación de entrada / salida.
- **Finalizado**: El programa ha terminado su ejecución.

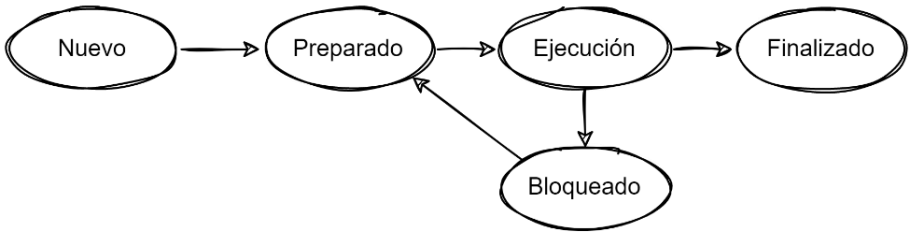


Figura 6: Estados de un proceso.

Hilos

Cuando se produce un cambio de contexto entre procesos, es decir, cuando se desaloja un proceso de la CPU para ejecutar otro diferente, el sistema operativo debe guardar, entre otras cosas, los valores de todos los registros de la CPU, información sobre el estado del proceso o los recursos utilizados, en el BCP del proceso y cargar los del nuevo proceso a ejecutar. Esta operación es costosa computacionalmente.

Además, cada proceso tiene su propio espacio de memoria principal reservado, sus propias variables y recursos asignados, por lo que compartir recursos, comunicar y sincronizar procesos no es una tarea sencilla o al menos todo lo sencilla que podría ser.

Los **hilos**, *threads* en inglés, procesos ligeros o también llamados subprocesos, solucionan estos problemas. Se tratan de secuencias de instrucciones de un mismo proceso que se pueden ejecutar de manera alternada o simultánea. Una de las diferencias con los procesos es que los procesos viven en el sistema operativo mientras que los hilos son partes de un proceso.

En la siguiente figura se puede ver una sencilla representación de la relación que existe entre procesos e hilos. Los procesos pueden tener de uno a múltiples hilos o subprocesos.

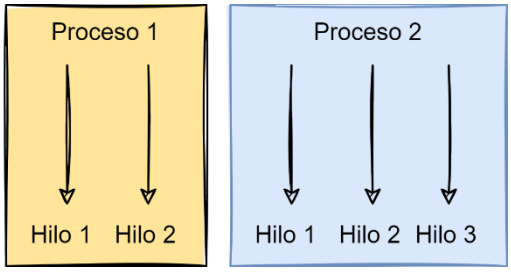


Figura 7: Procesos e hilos.

Los hilos no son entidades tan complejas como los procesos y por tanto se pueden gestionar de una forma más sencilla. Para empezar, los hilos de un mismo proceso comparten el mismo espacio de memoria, variables y recursos, por lo que el cambio de ejecutar un hilo a otro no es una operación tan costosa como el cambio de contexto entre procesos. Además, la comunicación entre hilos se simplifica, ya que se puede hacer a través de los espacios de memoria compartida.

Que un sistema sea multitarea permite que tengamos varios procesos abiertos. Permite, por ejemplo, que podamos escuchar música en *Spotify* mientras escribimos un documento en *Word* y el *Antivirus* hace un análisis del sistema.

Que un proceso tenga hilos permite que un mismo proceso pueda hacer varias tareas a la vez. Siguiendo los ejemplos anteriores, permite que *Spotify* reproduzca música mientras buscas las canciones de *J Balvin*, o que en el *Word* puedas escribir mientras el corrector ortográfico corrige las faltas.

Los hilos se utilizan mucho. Solo hay que pensar que todo programa que permita hacer más de una cosa a la vez hace uso de hilos. De hecho, cualquier programa con interfaz gráfica destina como mínimo un hilo solo para el dibujado de la interfaz y otro para toda la lógica del programa. En los siguientes capítulos nos centraremos en cómo programar utilizando hilos. De momento nos quedan unos pocos conceptos más que estudiar.

Programación concurrente

Vamos ahora con el concepto que da título a este libro. Según la *Wikipedia* el término **concurrente** hace referencia a la "habilidad de distintas partes de un programa, algoritmo, o problema de ser ejecutado en desorden o en orden parcial, sin afectar al resultado final". Así pues, programación concurrente hace referencia al hecho de desarrollar programas cuyas partes se pueden ejecutar de forma desordenada y esto se consigue mediante el uso de hilos.

Volvamos al ejemplo de la cocina. Imaginemos que nuestro cocinero (nuestro programa) quiere preparar un delicioso plato de pasta a la carbonara. Cada tarea representará un hilo: hervir la pasta, cortar la cebolla, batir las yemas, etc. El cocinero es capaz de hacer varias cosas a la vez, pero tiene que hacerlo con sentido. No debe mezclar las yemas con la pasta si ésta aún no está cocida. Aunque hay tareas que no requieren realizarse en un orden específico hay otras que sí.

El sistema operativo actúa de jefe, pero de jefe incompetente. Él es quien decide qué tarea se ejecuta en cada momento en la CPU. El problema es que el sistema operativo no sabe nada de cocina. No podemos culparle, tiene muchos trabajadores y no puede saber de todo. Él solo sabe que tiene un trabajador que quiere hacer varias tareas, pero no tiene ni idea de cuáles son ni en qué orden se deben realizar. Ante esta situación el sistema operativo no se complica la vida, ejecutará las tareas de forma desordenada a no ser que su trabajador le diga otra cosa.

Por eso, somos nosotros los que, al utilizar hilos, debemos utilizar mecanismos, llamados de comunicación y sincronización entre hilos, que aseguren que las partes importantes se ejecutan en el orden adecuado. El cocinero es el responsable de saber en qué orden debe realizar las tareas.

Como conclusión podemos definir la **programación concurrente** como el uso de técnicas para implementar algoritmos concurrentes, es decir, cuyas partes pueden ejecutarse de forma desordenada sin que esto afecte al resultado final. Como hemos dicho anteriormente, para ello se utilizan **mecanismos de comunicación** y **sincronización** que, por supuesto, estudiaremos más adelante.

Ventajas e inconvenientes

Las **ventajas** de la programación concurrente son:

- Se produce un **mayor aprovechamiento del uso de la CPU**, ya que mientras un proceso se encuentra bloqueado es posible ejecutar otro proceso.
- Conseguimos un **menor tiempo de ejecución**. En el caso de los sistemas multitarea se debe al mejor aprovechamiento del uso de la CPU. En el caso de los sistemas multiproceso se debe a que es posible ejecutar múltiples procesos de forma simultánea.
- **Facilita la resolución de ciertos problemas**. Imaginemos un servidor web que necesita responder a las peticiones de múltiples clientes. Si el tratamiento de las peticiones no fuera concurrente, el servidor no procesaría la última petición hasta haber respondido las anteriores.

Entre los inconvenientes encontramos solo uno, aunque bastante importante:

- La **programación es más compleja** ya que requiere la comunicación y sincronización de los procesos o hilos.

Aun así, las ventajas que aporta la programación concurrente supera con creces a sus inconvenientes.

Programación paralela y distribuida

Aquí tenemos otro par de conceptos que muchas veces se utilizan de forma indistinta cuando son cosas bien diferentes.

La **programación paralela** es un tipo de programación concurrente que se produce en los sistemas multiproceso donde múltiples procesos o hilos ejecutan sus instrucciones de forma simultánea. Esto es posible gracias al uso de varios procesadores o núcleos. Estos procesos trabajan de forma sincronizada para resolver una tarea compleja.

La **programación distribuida** es una forma de programación paralela que se basa en el uso de múltiples máquinas o nodos conectados a través de una red. De esta forma, es posible ejecutar de forma paralela los procesos, pero en máquinas que pueden estar ubicados en diferentes puntos. La comunicación en estos casos se suele llevar a cabo mediate *sockets*.

Escalabilidad vertical y horizontal

Si disponemos de una única máquina y los problemas que queremos resolver no dejan de aumentar en complejidad, necesitaremos mejorar sus recursos *hardware* (mejorar los procesadores, memoria *RAM*, etc.) para que tenga suficiente capacidad de cómputo.

Es lo mismo que ocurre cuando queremos jugar a un videojuego de última generación en un ordenador antiguo. Lo más seguro es que tengamos que comprar una nueva tarjeta gráfica, procesador o memoria *RAM*.

El hecho de aumentar la potencia de una máquina, mejorando o ampliando su *hardware* recibe el nombre de **escalado vertical**. Las ventajas de este tipo de escalado es que no tiene ninguna implicación sobre el *software*, que seguirá funcionando de la misma forma. El principal inconveniente es el incremento exponencial de los precios conforme mejoramos los componentes y que existe un límite tecnológico.

Una alternativa más económica y escalable es disponer de una red distribuida de ordenadores. Si queremos aumentar la capacidad de cómputo, solo será necesario añadir nuevos nodos. Este tipo de escalado recibe el nombre de **escalado horizontal**.

Su ventaja es que su capacidad de crecimiento es mayor, facilita la alta disponibilidad (si un nodo falla el sistema *no cae*, ya que el resto de los nodos siguen trabajando) y permite el balanceo de carga entre los nodos.

Entre sus inconvenientes encontramos que su configuración es más compleja y que requiere un *software* que coordine los diferentes nodos.

Programación Concurrente

Monitores

Definiciones

Sección crítica: Partes del código en las que se realizan operaciones sobre recursos compartidos.

Condición de carrera: Comportamiento del *software* por el que el resultado obtenido tras su ejecución depende del orden de ejecución de las operaciones concurrentes.

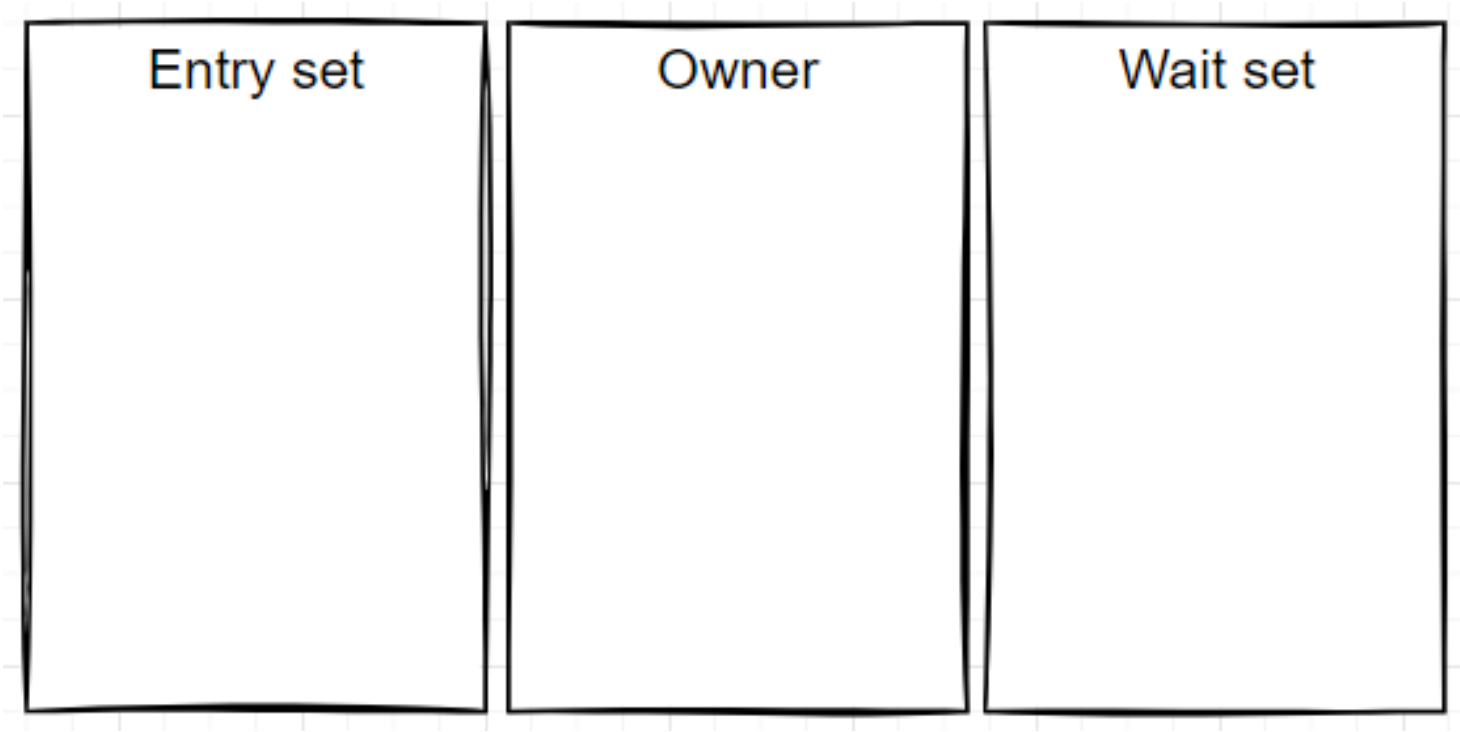
Monitor: Estructura diseñada para impedir que más de un hilo acceda a la sección o secciones críticas de un objeto y que se produzcan condiciones de carrera.

Monitor

Estructura diseñada para impedir que más de un hilo acceda a la sección o secciones críticas de un objeto.

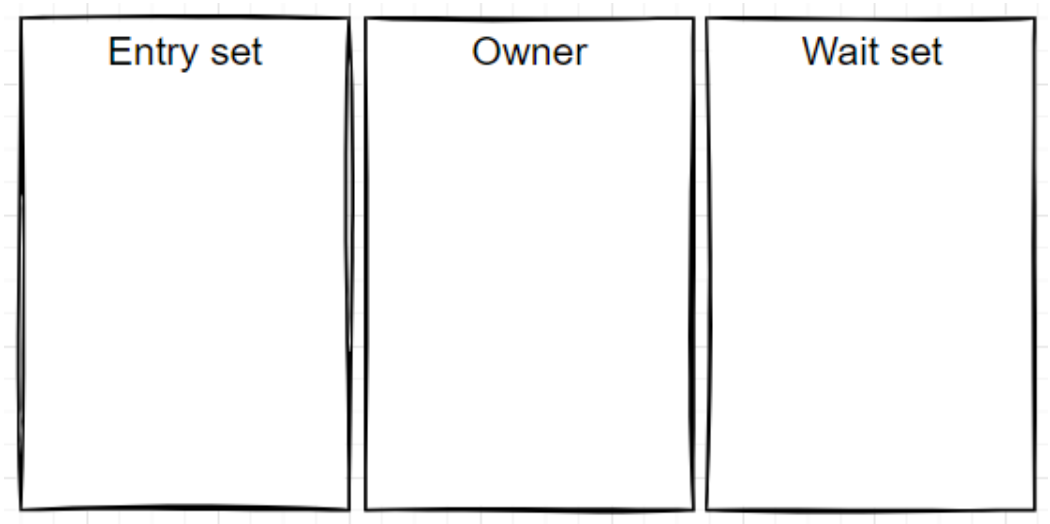
- En *Java* todos los objetos pueden actuar como monitor ➞ Mecanismo implementado en la clase *Object*.
- Podemos utilizar cualquier objeto para:
  - Sincronizar el acceso a las secciones críticas del propio objeto.
  - Sincronizar el acceso a las secciones críticas de otro objeto.

Monitor



Monitor

- Los hilos que se encuentren en el *Entry set* quieren acceder a una sección crítica protegida por el monitor.
- El hilo que sea el *Owner*, se dice que es el propietario del monitor. Este hilo podrá ejecutar las secciones críticas del objeto protegido.



Monitor

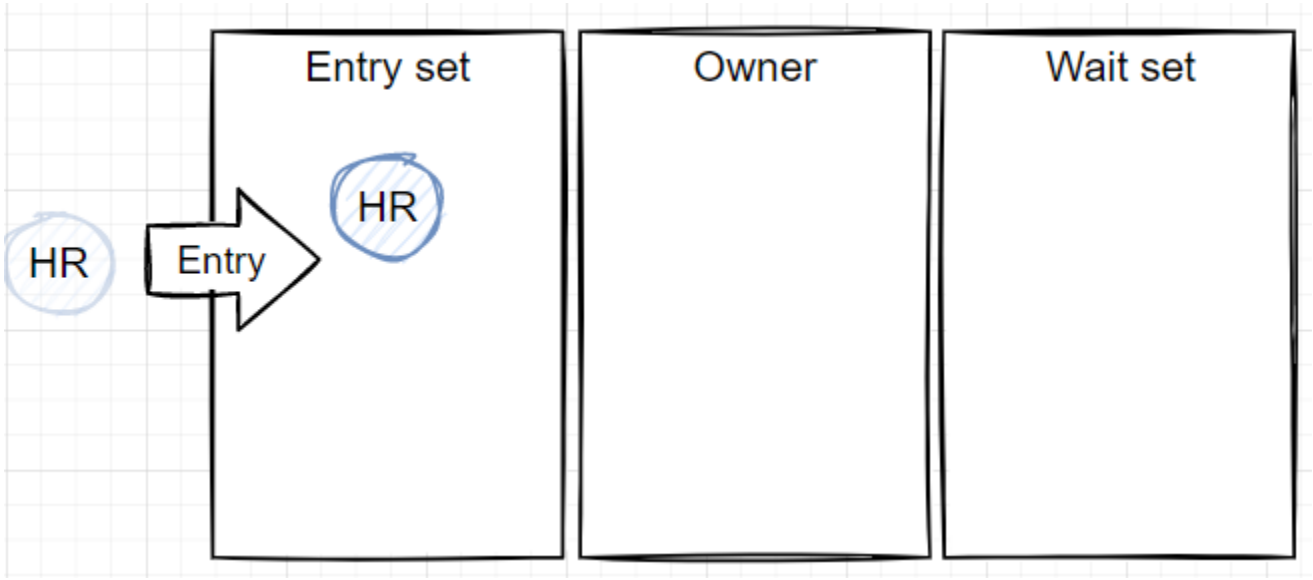
Ejemplo de la cuenta bancaria

- Hay dos secciones críticas: El método ingresar y el método reintegrar.
- Vamos a suponer que el objeto *cuenta*, que es el objeto compartido, actúa como monitor.
- En este caso el monitor del objeto *cuenta* protegerá el acceso a sus propias secciones críticas.

Monitor

Ejemplo de la cuenta bancaria

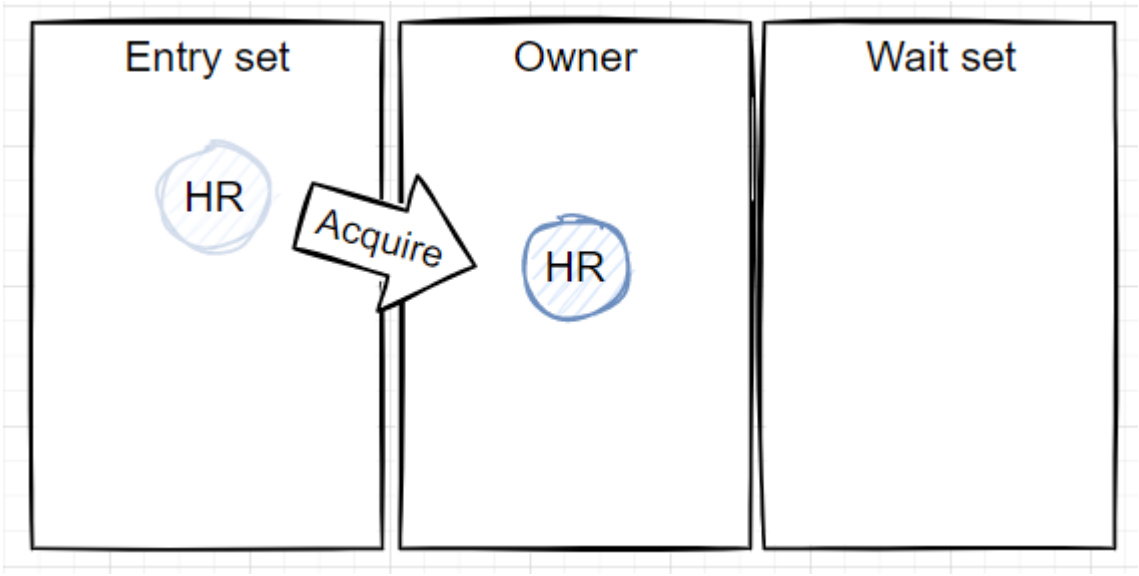
HiloReintegro quiere ejecutar el método *reintegrar*.



Monitor

Ejemplo de la cuenta bancaria

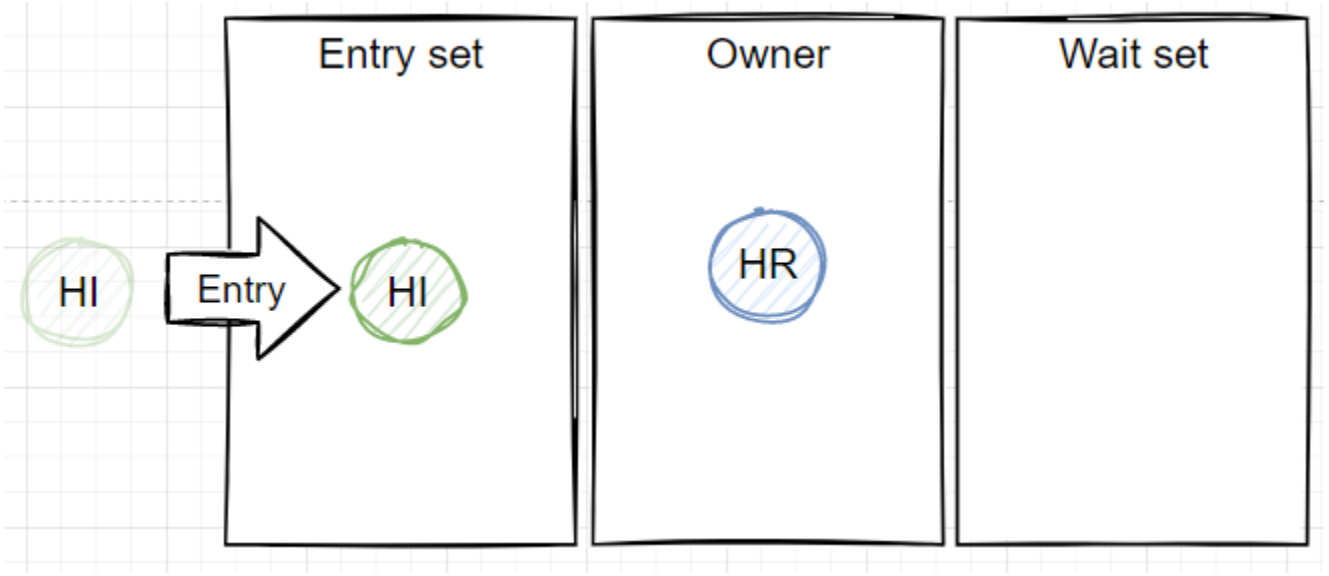
HiloReintegro adquiere la propiedad del monitor. Ahora puede ejecutar el método *reintegrar*.



Monitor

Ejemplo de la cuenta bancaria

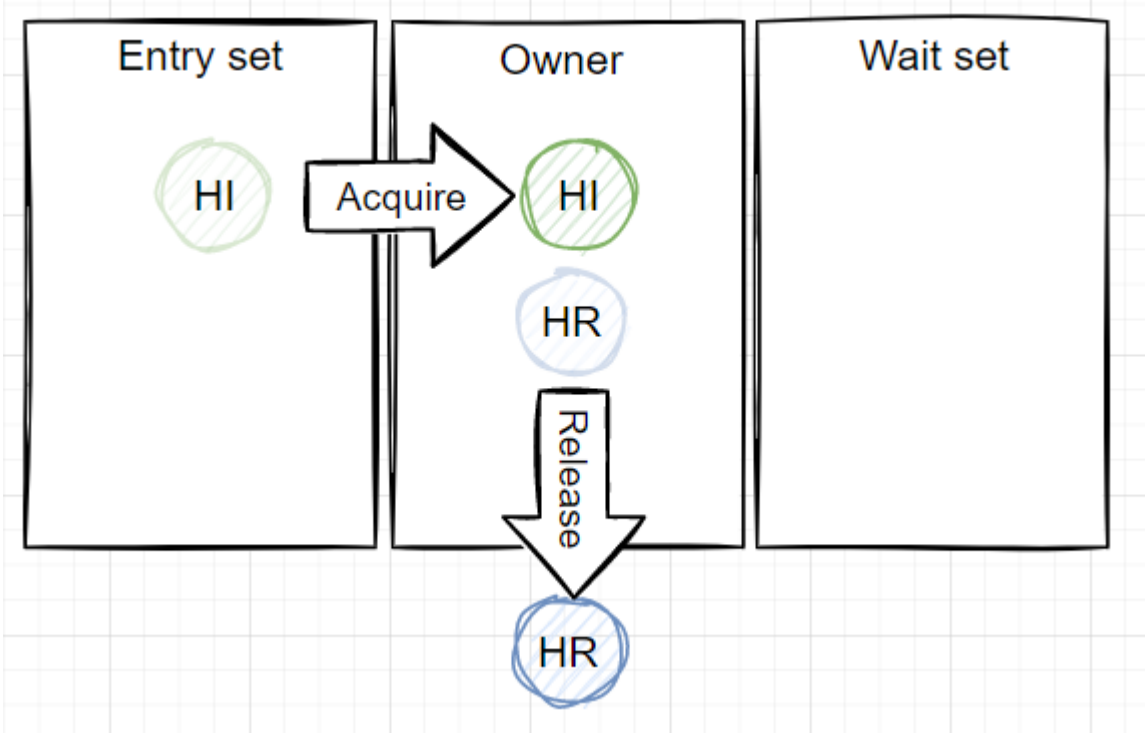
HiloIngreso quiere ejecutar el método *ingresar*. Llegará al *Entry set* pero no podrá adquirir la propiedad del monitor.



Monitor

Ejemplo de la cuenta bancaria

Cuando el propietario del monitor (*HiloReintegro*) termine de ejecutar la sección crítica, liberarará el monitor. Entonces *HiloIngreso podrá adquirirlo*.



Unión Europea
Fondo Social Europeo
El FSE invierte en tu futuro

IES Mare Nostrum

Programación Concurrente

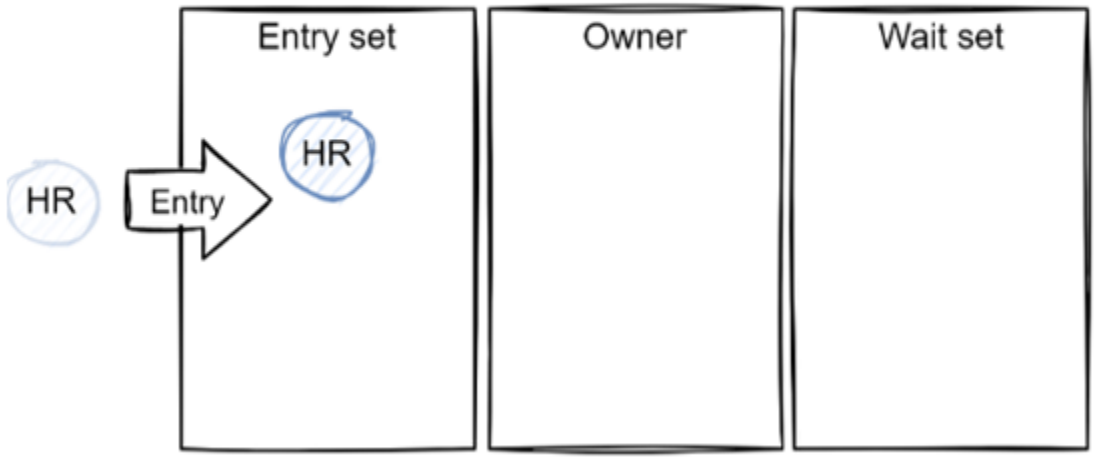
Métodos *notify* y *wait*

Métodos *notify* y *wait*

Hilo Reintegro

Comienza a ejecutarse *hilo reintegro*. Llega al bloque sincronizado, así que entra al *entry set* del monitor.

```
   for(int i = 0; i < 1000; i++) {
➤  synchronized (cuenta){
      while(cuenta.getSaldo() < 100){
          cuenta.wait();
       }
       cuenta.reintegrar(100);
   }}
```

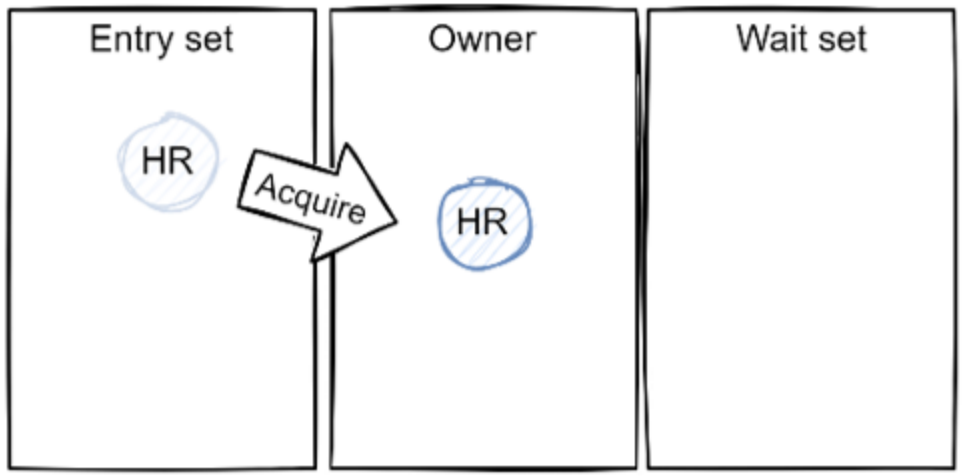


Métodos *notify* y *wait*

Hilo Reintegro

Puesto que el monitor no está bloqueado, el hilo puede entrar al bloque sincronizado, pasando a ser el propietario del monitor. Mientras el saldo sea superior o igual a 100€ realizará reintegros.

```
for(int i = 0; i < 1000; i++) {
  synchronized (cuenta){
   while(cuenta.getSaldo() < 100){
     cuenta.wait();
   }
   cuenta.reintegrar(100);
}}
```

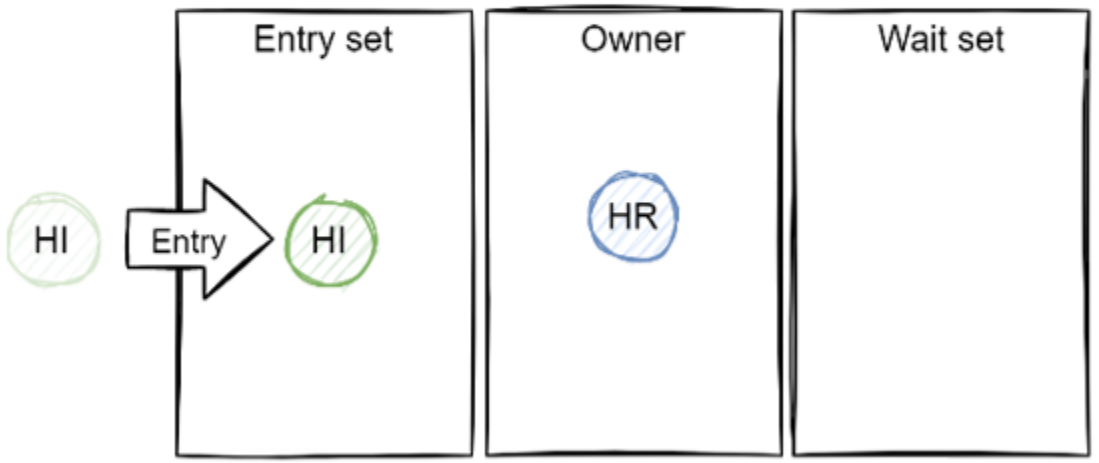


Métodos *notify* y *wait*

Hilo Ingreso

Supongamos que ahora pasa a ejecutarse *hilo ingreso*. Puesto que el monitor está bloqueado por *hilo reintegro*, quedará a la espera en *entry set* sin poder entrar al bloque sincronizado.

```
for(int i = 0; i < 1000; i++) {
➤     synchronized (cuenta){
          cuenta.ingresar(100);
          cuenta.notify();
      }
}
```

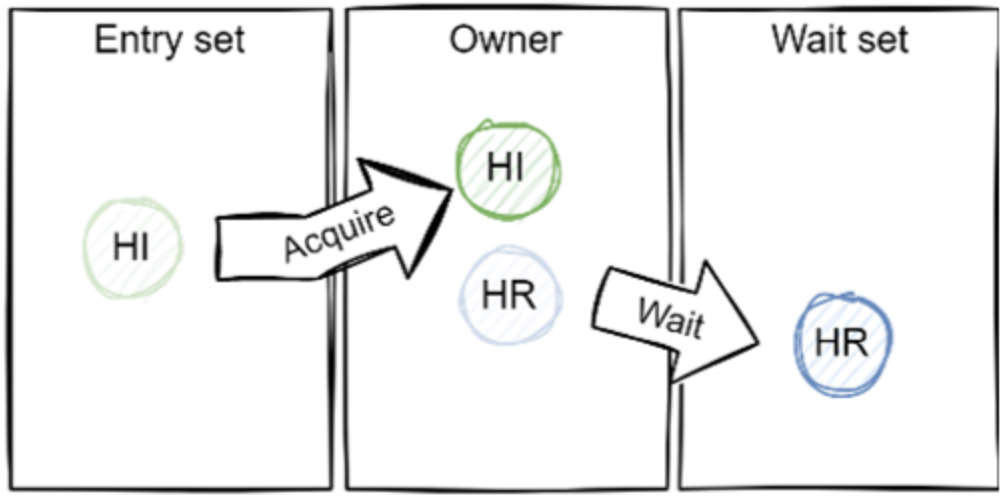


Métodos *notify* y *wait*

Hilo Reintegro

Hilo reintegro continúa su ejecución y llega un momento en el que el saldo es inferior a 100€, por lo que se cumple la condición del *while* y se ejecuta el método *wait*. En ese momento, el hilo pasará a la zona *wait set* donde permanecerá a la espera sin ejecutar ninguna línea más. Como consecuencia liberará el monitor e hilo ingreso tomará su posesión.

```
synchronized (cuenta){
    while(cuenta.getSaldo() < 100){
        cuenta.wait();
    }
    cuenta.reintegrar(100);
}
```

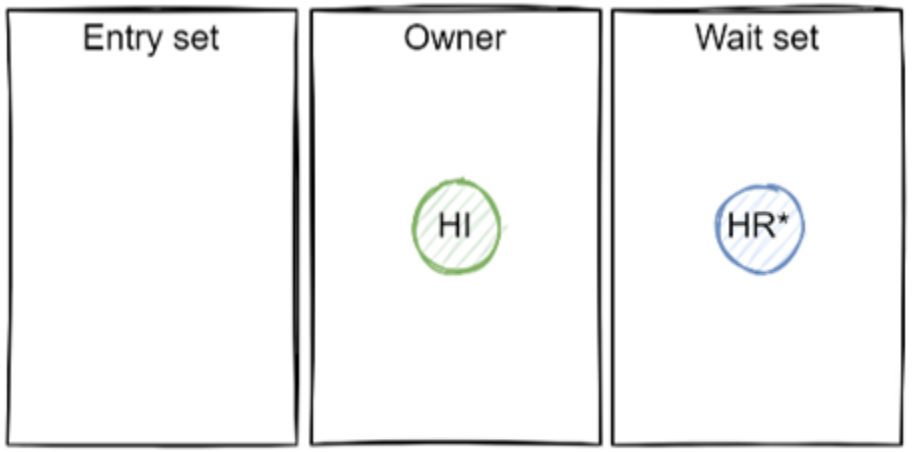


Métodos *notify* y *wait*

Hilo Ingreso

En este momento pasa a ejecutarse *hilo ingreso*. Tras realizar el ingreso, llama al método *notify* del monitor *cuenta*. Esto notificará a uno de los hilos que se encuentren en el *wait set* del monitor *cuenta*. En nuestro caso solo hay un hilo en el *wait set* por lo que será a ese al que se notificará. A partir de este momento, *hilo reintegro* permanecerá a la espera para adquirir la propiedad del monitor y así continuar su ejecución.

```
for(int i = 0; i < 1000; i++) {
    synchronized (cuenta){
        cuenta.ingresar(100);
        cuenta.notify();
    }
}
```



* Hilo notificado a la espera de adquirir el monitor.

Métodos *notify* y *wait*

A la hora de llamar a los métodos *notify* y *wait* hay que tener en cuenta:

- Las llamadas deben hacerse siempre desde dentro de bloques sincronizados. Si no lo hacemos, en tiempo de ejecución, cuando se llame a alguno de estos métodos, obtendremos el siguiente error:

```
Exception in thread java.lang.IllegalMonitorStateException:
current thread is not owner
```

- Las llamadas al método *wait* las haremos siempre desde dentro de un bucle *while* para evitar un efecto llamado *spurius wakups*.

Unión Europea
Fondo Social Europeo
El FSE invierte en tu futuro

IES Mare Nostrum

Programación Concurrente

Gestión de múltiples hilos

Introducción

Problema

- La creación de hilos es un proceso relativamente costoso.

Solución

- Interfaz *ExcutorService* ➞ gestión eficiente / reutilización de hilos.

ExecutorService

Crea un *pool* o conjunto de hilos que se reutilizan según se necesiten para ejecutar las diferentes tareas.

`void execute (Runnable command)`

Ejecuta en un hilo la tarea que recibe por parámetro.

`void shutdown()`

Hace que el executor no acepte más llamadas. El executor esperará a que terminen de ejecutarse las tareas que se encuentren en ejecución y las que estén a la espera de ejecutarse. Una vez finalizadas se cierra el executor.

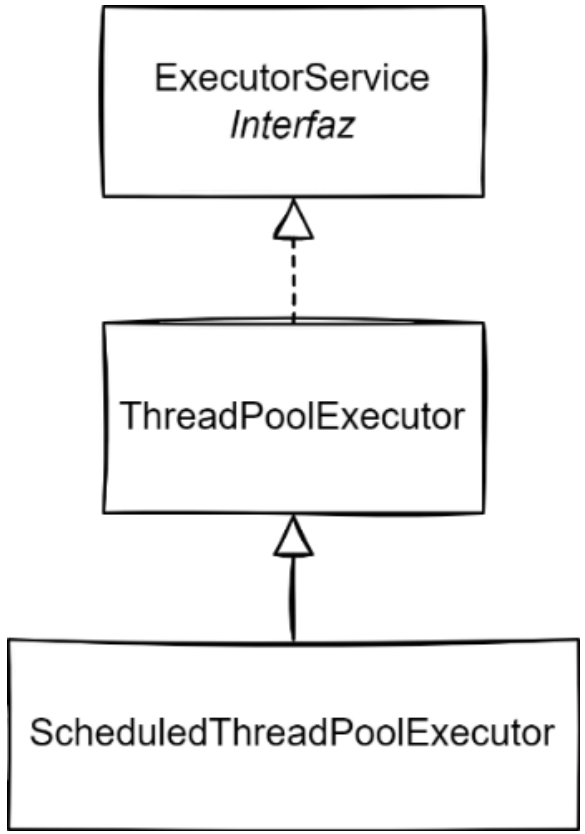
`List<Runnable> shutdownNow()`

Finaliza las tareas en ejecución de forma abrupta. Devuelve una lista con las tareas que estaban a la espera de ejecutarse.

`boolean awaitTermination(long timeout, TimeUnit unit)`

Llamada bloqueante que espera hasta que todas las tareas finalicen su ejecución o hasta que finalice la espera indicada en timeout. Este método se suele llamar tras llamar al método shutdown par esperar la finalización de las tareas.

ExecutorService



ThreadPoolExecutor

Implementa *ExecutorService*.

Constructores con un gran número de parámetros.

```
ThreadPoolExecutor(int corePoolSize, int maximumPoolSize, long keepAliveTime, TimeUnit unit, BlockingQueue<Runnable> workQueue)

ThreadPoolExecutor(int corePoolSize, int maximumPoolSize, long keepAliveTime, TimeUnit unit, BlockingQueue<Runnable> workQueue,
RejectedExecutionHandler handler)

ThreadPoolExecutor(int corePoolSize, int maximumPoolSize, long keepAliveTime, TimeUnit unit, BlockingQueue<Runnable> workQueue,
ThreadFactory threadFactory)

ThreadPoolExecutor(int corePoolSize, int maximumPoolSize, long keepAliveTime, TimeUnit unit, BlockingQueue<Runnable> workQueue,
ThreadFactory threadFactory, RejectedExecutionHandler handler)
```

Métodos que nos interesan:

`public int getPoolSize()`

Devuelve el número de hilos que existen en el pool.

`public int getActiveCount()`

Devuelve el número de hilos que están ejecutando alguna tarea.

ScheduledExecutorService

Hereda de *ThreadPoolExecutor*.

Constructores con un gran número de parámetros.

Métodos que nos interesan:

```
ScheduledFuture<?> schedule(Runnable command, long delay, TimeUnit unit)
```

Lanza una tarea cuando transcurra el tiempo indicado en *delay*. El parámetro *unit* indica la unidad de tiempo utilizada.

```
ScheduledFuture<?> scheduleAtFixedRate(Runnable command, long initialDelay,
                                       long period, TimeUnit unit)
```

Lanza una tarea una vez pasado el tiempo indicado en el parámetro *initialDelay* y la ejecutará repetidamente con la frecuencia indicada en el parámetro *period*.

Executors

Factory que facilita la instanciación de objetos de tipo *ThreadPoolExecutor* y *ScheduledThreadPoolExecutor*.

`static ExecutorService newFixedThreadPool(int nThreads)` — *Ejemplo 23 y 23b*

Crea un *thread pool* que reutiliza un número fijo de hilos *nThreads* con una cola de espera ilimitada.

`static ExecutorService newSingleThreadExecutor()` — *Ejemplo 24*

Crea un *thread pool* de un único hilo con una cola de espera ilimitada.

`static ExecutorService newCachedThreadPool()` — *Ejemplo 25*

Crea un *thread pool* que permite crear tantos hilos nuevos como resulten necesarios, reutilizando los que se hubiesen construido previamente si están disponibles.

Executors

Factory que facilita la instanciación de objetos de tipo *ThreadPoolExecutor* y *ScheduledThreadPoolExecutor*.

`static ScheduledExecutorService newScheduledThreadPool(int corePoolSize)` *Ejemplo 26 y 27*

Crea un *thread pool* de un número fijo de hilos *corePoolSize* hilos que se ejecutarán tras un determinado tiempo de espera o periódicamente.

`static ScheduledExecutorService newSingleThreadScheduledExecutor()`

Crea un *thread pool* que permite programar un único hilo que se ejecutará tras un determinado tiempo de espera o periódicamente

Cheatsheet

Thread

```
public Thread(Runnable target)
```

Instancia un objeto de tipo *Thread*. Recibe por parámetro un objeto que implemente la interfaz *Runnable*.

```
public static void sleep(long millis)
```

Detiene la ejecución del hilo durante *millis* milisegundos.

```
public void start()
```

Ejecuta el hilo. La Máquina Virtual de Java ejecuta el método *run* de este objeto.

```
public long threadId()
```

Devuelve el identificador único del hilo. Se trata de un número positivo generado en el momento que se crea el hilo.

```
public final String getName()
public final void setName(String name)
```

Permiten obtener y asignar un nombre al hilo.

```
public final int getPriority()
public final void setPriority(int newPriority)
```

Permiten obtener y fijar la prioridad del hilo. La prioridad es un valor entero entre 1 y 10, siendo 1 la prioridad mínima y 10 la máxima. Estos valores los encontramos en las constantes *Thread.MIN_PRIORITY* y *Thread.MAX_PRIORITY*.

```
public Thread.State getState()
```

Obtiene el estado del hilo, que puede ser:

- NEW: Hilo nuevo que aún no se ha lanzado con el método *start()*.
- RUNNABLE: Hilo que se encuentra en ejecución. Esto es, que sus instrucciones se están ejecutando en el procesador.
- BLOCKED: Hilo que permanece bloqueado al tratar de acceder a un recurso ya ocupado.
- WAITING: Hilo que permanece a la espera indefinidamente hasta que se produzca un evento. El hilo permanecerá en este estado al llamar a los métodos *wait* y *join* en las versiones sin *timeout*.
- TIMED_WAITING: Hilo que permanece a la espera una cantidad determinada de tiempo. Cuando un hilo ejecuta el método *sleep* pasa a estar en este estado. También al llamar a los métodos *wait* y *join* cuando se utiliza un *timeout*.
- TERMINATED: Hilo finalizado.

El estado devuelto por *getState()* es un enumerado de tipo *Thread.State*.

```
public final boolean isAlive()
```

Devuelve si el hilo aún está vivo o no, es decir, si ya ha terminado su ejecución.

```
public static Thread currentThread()
```

Devuelve una referencia al hilo que actualmente se está ejecutando.

```
public final void join() throws InterruptedException
```

Espera a que el hilo finalice. Se trata de una llamada bloqueante.

```
void interrupt()
```

Interrumpe el hilo actual.

```
boolean isInterrupted()
```

Devuelve si el hilo actual ha sido interrumpido.

Object

```
void  wait()
void  wait(long timeout)
```

Hace que el hilo actual se suspenda a la espera de que otro hilo invoque al método *notify* o *notifyAll* de este objeto.

```
void notify()
```

Despierta a uno de los hilos suspendidos a la espera del monitor de este objeto. Este método solo debe ser llamado por un hilo que sea el propietario del monitor del objeto.

```
void notifyAll()
```

Despierta a todos los hilos suspendidos a la espera del monitor de este objeto. Este método solo debe ser llamado por un hilo que sea el propietario del monitor del objeto.

ExecutorService (interfaz)

```
void execute (Runnable command)
```

Ejecuta en un hilo la tarea que recibe por parámetro.

```
void shutdown()
```

Hace que el *executor* no acepte más llamadas. El *executor* esperará a que terminen de ejecutarse las tareas que se encuentren en ejecución y las que estén a la espera de ejecutarse. Una vez finalizadas se cierra el *executor*.

```
List<Runnable> shutdownNow()
```

Finaliza las tareas en ejecución de forma abrupta. Devuelve una lista con las tareas que estaban a la espera de ejecutarse.

```
boolean awaitTermination(long timeout, TimeUnit unit)
```

Llamada bloqueante que espera hasta que todas las tareas finalicen su ejecución o hasta que finalice la espera indicada en *timeout*. Este método se suele llamar tras llamar al método *shutdown* par esperar la finalización de las tareas.

ThreadPoolExecutor

Implements *ExecutorService*

```
public int getPoolSize()
```

Devuelve el número de hilos que existen en el *pool*.

```
public int getPoolSize()
```

Devuelve el número de hilos que existen en el *pool*.

```
public int getActiveCount()
```

Devuelve el número de hilos que están ejecutando alguna tarea.

ScheduledExecutorService

Hereda de *ThreadPoolExecutor*

```
ScheduledFuture<?> schedule(Runnable command, long delay,
                            TimeUnit unit)
```

Lanza una tarea cuando transcurra el tiempo indicado en *delay*. El parámetro *unit* indica la unidad de tiempo utilizada.

```
ScheduledFuture<?> scheduleAtFixedRate(Runnable command,
                       long initialDelay, long period, TimeUnit unit)
```

Lanza una tarea una vez pasado el tiempo indicado en el parámetro *initialDelay* y la ejecutará repetidamente con la frecuencia indicada en el parámetro *period*.

Executors

```
static ExecutorService newFixedThreadPool(int nThreads)
```

Crea un *thread pool* que reutiliza un número fijo de hilos *nThreads* con una cola de espera ilimitada.

```
static ExecutorService newSingleThreadExecutor()
```

Crea un *thread pool* de un único hilo con una cola de espera ilimitada.

```
static ExecutorService newCachedThreadPool()
```

Crea un *thread pool* que permite crear tantos hilos nuevos como resulten necesarios, reutilizando los que se hubiesen construido previamente si están disponibles.

```
static ScheduledExecutorService
            newScheduledThreadPool(int corePoolSize)
```

Crea un *thread pool* de un número fijo de hilos *corePoolSize* hilos que se ejecutarán tras un determinado tiempo de espera o periódicamente.

```
static ScheduledExecutorService newSingleThreadScheduledExecutor()
```

Crea un *thread pool* que permite programar un único hilo que se ejecutará tras un determinado tiempo de espera o periódicamente.

MODELO Examen Programación Servicios y Procesos – Trimestre 1 – 2º DAM O/U

NOMBRE Y APELLIDOS:

Ejercicio 1 (4 puntos)

Responde a las siguientes preguntas tipo test. Dos respuestas incorrectas restan una correcta. Una respuesta incorrecta resta media correcta.

1. Un sistema informático tiene instalado un microprocesador con múltiples núcleos. Podemos decir que:
    a. Se trata de un sistema multiproceso.
    b. Se trata de un sistema monoproceso.
    c. Ninguna de las opciones es correcta.
2. La técnica denominada multiprogramación permite:
    a. Ejecutar múltiples procesos de forma intercalada.
    b. Ejecutar múltiples procesos de forma simultánea.
    c. Las otras dos opciones son correctas.
3. ¿Cuál es la función del Bloque de Control de Proceso (BCP)?
    a. Almacenar datos en disco duro.
    b. Almacenar información relativa al proceso.
    c. Bloquear los procesos con el fin de controlar su acceso.
4. ¿Qué ventajas tiene la programación concurrente?
    a. Facilita la programación de algoritmos.
    b. Se produce un mayor aprovechamiento de la CPU.
    c. Las otras respuestas son correctas.
5. Para lanzar o ejecutar un hilo llamaremos al siguiente método de la clase *Thread*:
    a. *start*.
    b. *launch*.
    c. *run*.
6. La diferencia entre implementar la interfaz *Runnable* y heredar de la clase *Thread* es que:
    a. Si implementamos la interfaz *Runnable* aún podemos heredar de otra clase.
    b. Si heredamos de la clase *Thread* no podemos implementar ninguna interfaz.
    c. No existe ninguna diferencia.
7. El hilo principal *M* ejecuta las siguientes líneas de código:

```
Thread h = new Thread(new H());
h.start();
Thread t = Thread.currentThread();
System.out.println("ID: " + t.getId());
System.out.println("Nombre: " + t.getName());
```

    ¿A qué hilo pertenece la información que se está mostrando por pantalla?
    a. Al hilo principal *M*.
    b. Al hilo *H*.
    c. A ninguno de los dos.

MODELO Examen Programación Servicios y Procesos – Trimestre 1 – 2º DAM O/U

NOMBRE Y APELLIDOS:

8. El método *interrupt*:
    a. Pertenece a la clase *Thread*.
    b. Debemos implementarlo si queremos utilizarlo para detener un hilo.
    c. Pertenece a la interfaz *Runnable*.
9. Si al llamar al método *interrupt* de un hilo éste se encuentra suspendido por estar ejecutando el método *sleep*:
    a. El método *sleep* lanzará una excepción.
    b. El hilo no se interrumpirá hasta que no finalice la llamada *sleep*.
    c. El hilo no se interrumpirá en ningún caso.
10. Si utilizamos *flags* para finalizar un hilo:
    a. No necesitamos esperar su finalización mediante el método *join* ya que finalizará de manera instantánea.
    b. La ejecución del hilo continuará hasta que se compruebe el valor del *flag*.
    c. Es un mecanismo en desuso en favor de las interrupciones.
11. ¿Qué técnica utilizada en clase permite a diferenes hilos comunicarse (intercambiar información) entre sí?
    a. Uso de memoria compartida.
    b. Comunicación interprocesos IPC.
    c. Envío de mensajes asíncronos.
12. Las secciones críticas:
    a. Son aquellas partes del código en las que accedemos y manipulamos recursos compartidos.
    b. Son mecanismos que provee *Java* para compartir recursos.
    c. Son mecanismos que provee *Java* para evitar condiciones de carrera.
13. ¿Qué podemos afirmar dado el siguente fragmento de código?

```
public static void main(String[] args) {
    Contador c;
    for(int i = 1; i <= 5; i++){
        c = new Contador();
        Thread t = new Thread(c);
        t.setName("Hilo " + i);
        t.start();
    }
}
```

    a. Todos los hilos creados comparten un mismo objeto contador.
    b. Cada hilo creado tiene su propio objeto contador.
    c. Los hilos están compartiendo memoria ya que existe un objeto compartido.
14. ¿Para qué sirve el mecanismo denominado monitor?
    a. Para evitar que varios hilos accedan a una misma sección crítica con el objetivo de que así se puedan producir condiciones de carrera.

NOMBRE Y APELLIDOS:

b. Para evitar que varios hilos accedan a una misma sección crítica con el objetivo de que no se produzcan condiciones de carrera.
c. Para que varios hilos puedan acceder a una misma sección crítica y así se puedan producir condiciones de carrera.

15. Dadas estas dos clases:

```
public class HiloIngreso implements Runnable {

   private Cuenta cuenta;

   public HiloIngreso(Cuenta cuenta) {
      this.cuenta = cuenta;
   }

   @Override
   public void run() {
      for(int i = 0; i < 1000; i++) {
         synchronized (cuenta){
            cuenta.ingresar(100);
         }
      }
   }
}

public class HiloReintegro implements Runnable {

   private Cuenta cuenta;

   public HiloReintegro(Cuenta cuenta) {
      this.cuenta = cuenta;
   }
   @Override
   public void run() {
      for(int i = 0; i < 1000; i++) {
         synchronized (cuenta){
            cuenta.reintegrar(100);
         }
      }
   }
}
```

¿Qué podemos afirmar?

a. Nunca se podrá ejecutar el método *ingresar* si se está ejecutando el método *reintegrar*.
b. Se podrá ejecutar el método *reintegrar* aunque se esté ejecutando el método *ingresar*.
c. No tenemos suficiente información para poder asegurar qué ocurrirá.

16. El hilo H_1 ha llamado al método *wait* del objeto *p*. Al hacerlo se ha suspendido su ejecución. Se reanudará cuando:

a. Cualquier otro hilo llame al método *notify* del objeto *p*.
b. El hilo H_1 llame al método *notify* del objeto *p*.
c. El objeto *p* llame al método *notify* del hilo H_1.

MODELO Examen Programación Servicios y Procesos – Trimestre 1 – 2º DAM O/U

NOMBRE Y APELLIDOS:

17. ¿Qué afirmación es cierta sobre el método *shutdown* de un *ExecutorService*?
    a. Finaliza de forma abrupta todos los hilos del *thread pool*.
    b. Es una llamada bloqueante, de forma que el hilo invocante quedará suspendido hasta que todos los hilos finalicen.
    c. Es una llamada no bloqueante que indica al *ExecutorService* que no debe aceptar más tareas.
18. Ejecutamos tres tareas *T1*, *T2* y *T3* utilizando un *thread pool*. Tras realizar múltiples ejecuciones observamos que al principio se suelen ejecutar dos de las tres tareas, pero la tercera no se ejecuta hasta que una de las dos anteriores finaliza. Podemos afirmar que:
    a. Estamos utilizando un *thread pool* de tamaño 1.
    b. Estamos utilizando un *thread pool* de tamaño 2.
    c. Estamos utilizando un *thread pool* de tipo *cached*.
19. Si utilizamos la palabra reservada *async* a la hora de definir un método…
    a. Éste método se ejecutará asíncronamente haciendo uso de un hilo.
    b. Podremos utilizar el operador *await* en el cuerpo del método.
    c. La ejecución del método pausará la ejecución del hilo principal hasta su finalización.
20. Tenemos un método cuya firma es:

```
async void LeerAchivo()
```

    Desde el método *Main* queremos llamar a *LeerArchivo* de la siguiente forma:

```
await LeerAchivo()
```

    ¿Cuál de las siguientes afirmaciones es cierta?
    a. Para poder realizar la operación *await* es necesario que el método *LeerArchivo* devuelva un objeto de tipo *Task*.
    b. Para declarar el método LeerArchivo como *async* es necesario que devuelva un objeto de tipo *Task*.
    c. El código anterior es correcto.
21. ¿Cuál es la función principal del nivel de transporte en el modelo TCP/IP?
    a. Permitir que un datagrama llegue desde un dispositivo origen a otro en cualquier parte del mundo.
    b. Entregar los datos a la aplicación correspondiente y distinguir entre diferentes aplicaciones mediante el uso de puertos.
    c. Transportar datos entre dispositivos que se encuentran dentro de una misma red.
22. ¿Qué protocolo del nivel de transporte no asegura que los paquetes lleguen de forma ordenada y sin errores?
    a. IP.
    b. UDP.
    c. TCP.

MODELO Examen Programación Servicios y Procesos – Trimestre 1 – 2º DAM O/U

NOMBRE Y APELLIDOS:

23. ¿Qué función principal realiza el protocolo DHCP?
    a. Asignar automáticamente direcciones IP en una red.
    b. Enviar correos electrónicos a través de una red.
    c. Convertir nombres de dominio en direcciones IP.
24. Para que un *socket* cliente se conecte a un *socket* servidor ¿Qué necesita conocer?
    a. La IP del servidor y el puerto en el que escucha el *socket* servidor.
    b. Únicamente la IP del servidor.
    c. El puerto del *socket* cliente.
25. ¿Cuál de las siguientes afirmaciones es cierta?
    a. Las clases *DataInputStream* y *DataOutputStream* únicamente disponen de métodos para leer y escribir *bytes*.
    b. Las clases *BufferedInputStream* y *BufferedOutputStream* disponen de métodos para leer y escribir diferentes tipos de datos tales como enteros, *doubles*, cadenas de texto, etc.
    c. Las clases *ObjectInputStream* y *ObjectOutputStream* disponen de métodos para leer y escribir instancias de objetos.

HOJA DE RESPUESTAS

1		11		21	
2		12		22	
3		13		23	
4		14		24	
5		15		25	
6		16			
7		17			
8		18			
9		19			
10		20			