Concurrencia y programación orientada a objeto

La concurrencia representa un concepto relativamente nuevo en los lenguajes de programación. Antes de la aparición de este concepto, se usaban las interrupciones del sistema operativo, a bajo nivel, para lograr comportamiento concurrente. Esto significa que lograr un programa concurrente era difícil de desarrollar, y además, al ser dependiente del SO, no era portable. Actualmente, lenguajes como Java, C#, Object Pascal y otros la implementan, logrando que el programador se desentienda de la cantidad de procesadores que tiene la máquina o de la administración de los procesos.

La concurrencia, antes reservada para ciertos problemas muy específicos y alejada de las aplicaciones corrientes, está teniendo un uso cada vez mayor por la proliferación de ambientes multitarea, las arquitecturas cliente-servidor y las aplicaciones distribuidas.

Un ejemplo clásico de concurrencia es la descarga de archivos extensos, como clips de audio o video, desde la web. Los usuarios no desean esperar hasta que se descargue todo un clip completo para empezar a reproducirlo. Para resolver este problema se pueden poner varios subprocesos a trabajar; un subproceso descarga un clip y el otro lo reproduce. Estas actividades o tareas se llevan a cabo concurrentemente. Para evitar que la reproducción del clip tenga interrupciones, se sincronizan los subprocesos de manera que el subproceso de reproducción no empiece sino hasta que haya una cantidad suficiente de clip en memoria, como para mantener ocupado al subproceso de reproducción.

Otro caso: en muchas páginas Web se puede desplazar la página e ir leyendo el texto antes de que todas las imágenes estén presentes en la pantalla. En este caso, el navegador está trayéndose en una tarea las imágenes, y soportando el desplazamiento de la página en otra tarea diferente.

Este concepto ha despertado el interés en el área informática, dado que implica grandes posibilidades, que tienen que ver con el mejor uso de los recursos de computación. Desde los años ’80 se ha incursionado en este tema, y si bien se ha divagado y entrado muchas veces en el terreno de la fantasía, en todas las áreas los proyectos del pasado han sido los puntos de partida para importantes innovaciones que hoy en día se están comenzando a poner en práctica.

Conceptos generales

Haciendo varias cosas a la vez

El concepto de concurrencia tiene que ver con hacer más de una cosa al mismo tiempo.

Los seres humanos frecuentemente realizan varias tareas al mismo tiempo. Por ejemplo: al caminar, simultáneamente el corazón bombea sangre. Mas allá de que es una tarea inconsciente, puede decirse que el cerebro atiende ambas tareas a la vez. Un ejemplo similar es mirar televisión mientras se prepara la comida, aunque en este caso lo que se hace en realidad es alternar las tareas, usando los tiempos libres que deja una de ellas, probablemente prioritaria, para atender a la otra. También puede darse el caso de que realmente se realicen dos cosas activamente, como escuchar música y escribir una carta. En otras ocasiones, en cambio, se cuenta con la ayuda de un colaborador, que permite avanzar con una tarea mientras se hace otra, necesaria para la finalización de aquélla, como por ejemplo un cirujano, que trabaja con un instrumentista.

Circunstancias similares ocurren en el área informática. En algunas ocasiones, un programa aparentemente tiene el monopolio del procesador, pero hay un sistema operativo que corre tras él. Hay casos en que varios programas corren alternándose en el uso del único procesador. A veces pueden correr varios programas realmente en paralelo sobre una máquina de múltiples procesadores. Y en el caso más amplio, varias computadoras pueden trabajar en conjunto para resolver un determinado problema.

Un caso simple de concurrencia se da en las interfaces de usuario. Por ejemplo, mientras el sistema está haciendo una tarea se puede presionar un botón de cancelación, y ese evento va a ser atendido simultáneamente.

Concurrencia

Concurrencia es la técnica de programación que permite que un programa haga más de una cosa a la vez, en una misma o en varias computadoras. Por ejemplo, si en una aplicación de facturación se necesita acceder a un archivo para obtener la descripción de un producto y a otro para obtener el precio, no necesariamente se debe hacer una cosa después de la otra. Si se realiza de esta manera puede ser porque el

lenguaje o la plataforma no permiten resolverlo de otro modo, pero más allá de esas limitaciones, conceptualmente se debería poder hacer en paralelo.

Sin embargo, la concurrencia a menudo no es más que una ilusión generada por el sistema informático, ya que en máquinas de un solo procesador no suele existir una manera de que varias tareas sean hechas al mismo tiempo. En este sentido, se pueden distinguir tres tipos de concurrencia:

- Multiprocesamiento
- Multiprogramación
- Computación distribuida

Multiprocesamiento es la técnica mediante la cual *varios procesos usan varios procesadores*, a razón de por lo menos un procesador por proceso. Cada proceso ocupa un espacio de memoria independiente (para no interferir con la ejecución de otros procesos). Es una técnica útil para resolver problemas de cálculo intensivo, aunque sólo es utilizable en máquinas de múltiples procesadores y en ciertos sistemas operativos.

Multiprogramación, en cambio, es la técnica que se da cuando *el número de procesadores es menor que el número de procesos*. Habitualmente se dan varios procesos corriendo sobre un único procesador, coordinados por un algoritmo que asigna recursos a los distintos procesos.

Computación distribuida hace referencia al hecho de correr programas diferentes en varias computadoras que se envían mensajes y comparten objetos. Este tipo de sistemas se compone, básicamente, de distintas máquinas situadas de forma dispersa, que interaccionan entre sí, es decir, se comunican y cooperan con el objetivo de realizar, de un modo eficiente y fiable, la tarea encomendada a todo el conjunto. Si bien no se la considera concurrencia en sentido estricto, se trata de un caso particular.

A medida que se va hacia esquemas más distribuidos se logra una mayor distribución del trabajo. Es decir, el multiprocesamiento permite distribuir mejor que la multiprogramación y la computación distribuida mejor que aquél. No obstante, esto no implica necesariamente una mayor eficiencia. El rendimiento puede medirse en función del tiempo de respuesta del sistema o de la cantidad de trabajo que realiza por una unidad de tiempo (*troughput*). Si bien hay una cierta correlación entre descentralización y cantidad de trabajo por unidad, la descentralización implica mayores necesidades de comunicación. Este no es un tema trivial, ya que puede significar inconvenientes, y es especialmente crítico en las aplicaciones distribuidas, por lo que, en general se busca minimizar la comunicación entre los distintos procesos.

Los siguientes aspectos de la concurrencia hacen que su uso requiera un estudio minucioso:

- Las tareas compiten por los recursos (todo lo que necesita un proceso para operar), como espacio de memoria, dispositivos de entrada/salida, etc. Se debe garantizar que todas las tareas puedan acceder a los recursos necesarios para seguir procesando en un tiempo finito.
- Desde el punto de vista de los tipos de métodos, existe una diferencia entre las operaciones de consulta provenientes de distintas tareas y las operaciones de modificación. Las primeras pueden hacerse en cualquier orden sobre el mismo objeto, mientras que en las segundas, el orden de las operaciones puede alterar el resultado.

Procesos e hilos

Una forma habitual de la concurrencia es la que consiste en correr *varios procesos* en forma simultánea en una misma computadora. Se llama **proceso** a un programa en ejecución. Por lo tanto, un proceso es una entidad activa, que ocupa recursos y tiene asignado un espacio de direcciones de memoria, una serie de archivos abiertos, registros, etc.

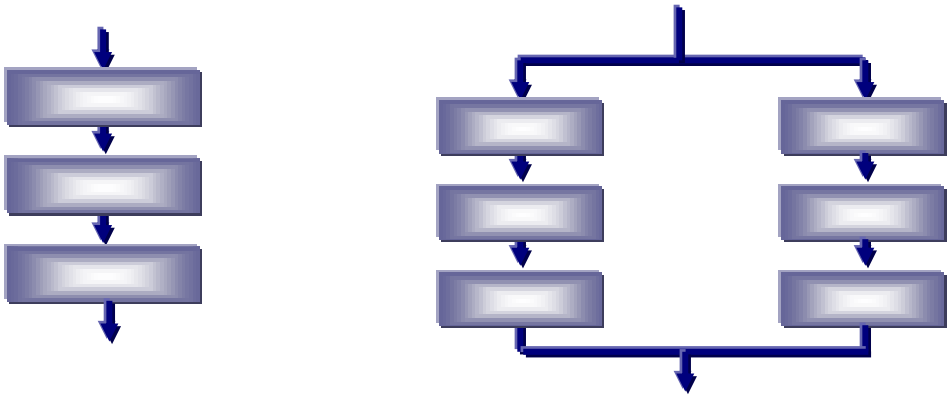
En general, en este caso no se permite que un proceso acceda a la partición de memoria de otro proceso.

Esta modalidad es manejada por el sistema operativo. Este debe hacer complejas planificaciones de distribución de tiempos. A pesar de que la sobrecarga en recursos del sistema es menor que en otras soluciones, puede haber inconvenientes cuando un proceso cause la caída de todos los demás por una falla que se transfiere al sistema operativo. Además, la comunicación entre procesos es bastante costosa.

Una variante más liviana, muy usada hoy en día es la llamada **concurrencia con múltiples hilos**, también conocida como multihilos ("multithreading").

En este caso, un solo programa, es decir, *un solo proceso*, posee hilos que se pueden ejecutar en forma paralela. Los recursos los posee el proceso, y un hilo puede acceder a cualquier dirección, incluyendo

memoria, archivos abiertos y dispositivos, y modificar objetos de otro hilo. De este modo, la comunicación entre hilos es muy eficiente.



Programa de Flujo único (Singlethread) Programa de Flujo múltiple (Multithread)

En un contexto de múltiples hilos:

- Los hilos pueden ejecutarse en un único procesador o soportar el concepto de multiprocesamiento en entornos de varios procesadores.
- Los hilos pueden competir abiertamente por los recursos o ser cuidadosamente puestos en una planificación.
- Los hilos pueden ser bloqueados por otros.

Es habitual que concurrencia se use como sinónimo de multihilos. Esto se debe a que Java, como otros lenguajes de programación, tiene soporte para hilos, independientemente de la plataforma.

Concurrencia y lenguajes de programación

En POO el énfasis se pone más en los objetos que en las actividades, lo cual puede parecer que resta importancia a la forma en que se ejecutan las distintas tareas. Sin embargo, la propia definición de un sistema como un conjunto de objetos autónomos colaborando, es más bien activa y concurrente.

Es decir, al pensar en objetos recibiendo mensajes y respondiendo a esos estímulos parecería un modelo de objetos pasivos, en el cual no se presta demasiada atención a las actividades. Sin embargo, esto no implica que estos objetos no reciban esos estímulos de tareas que se estén ejecutando concurrentemente, y que un mismo objeto pueda estar recibiendo mensajes provenientes de varios objetos diferentes a la vez.

Pero incluso se puede pensar en objetos activos, que manejan actividades por sí mismos y disparan otras actividades en forma concurrente. Por añadidura, estos objetos activos podrían residir en máquinas diferentes. Los hilos son un caso particular de objeto activo.

De allí que la POO, no sólo no esté en contradicción con la programación concurrente, sino que incluso la enfatiza. Tal vez por eso haya soporte a la concurrencia en los primeros lenguajes de programación basados en objetos. La propia idea de objetos pasándose mensajes es concurrente.

Adicionalmente, la POO mejora el uso de concurrencia gracias al encapsulamiento, la modularidad y la extensibilidad. Varios lenguajes orientados a objetos, como Java, soportan la concurrencia sin necesidad de recurrir a bibliotecas, lo que permite mayores comprobaciones y optimizaciones en tiempo de compilación.

Como en tantas cosas, fue Ada el precursor en el manejo de multihilos a alto nivel.

En los lenguajes que manejan multihilos con un adecuado nivel de abstracción, el programador no se debe preocupar por el orden en que se ejecutan los hilos, o más bien, debe saber que no puede confiar en que los hilos se ejecuten en un determinado orden, pues el procesador los atenderá en la forma en que mejor le parezca. Si no se maneja adecuadamente, esto puede ser más un problema que una ventaja.

Hilos en Java

Los hilos o threads en Java proveen los mecanismos que permiten compartir recursos a través de prioridades, de forma sincronizada, cuando fuera necesario. La propia Máquina Virtual Java (JVM) es un

sistema multi-thread. Es decir, es capaz de ejecutar varias secuencias de ejecución (programas) simultáneamente. La JVM gestiona todos la asignación de tiempos de ejecución, prioridades, etc, de forma similar a como gestiona un Sistema Operativo (SO) múltiples procesos. La diferencia básica entre un proceso de SO y un thread Java es que los threads corren dentro de la JVM, que es **un** proceso del SO, y por tanto comparten todos los recursos, incluida la memoria y las variables y objetos allí definidos. Este tipo de procesos donde se comparten los recursos se conoce como 'proceso ligero' (lightweight process).

Java da soporte al concepto de hilo desde el mismo lenguaje, con clases e interfaces definidas en el package java.lang y con métodos específicos para su manipulación en la clase Object.

Desde el punto de vista de las aplicaciones los hilos son útiles ya que permiten que el flujo del programa sea divido en dos o más partes, cada una ocupándose de alguna tarea. Por ejemplo un hilo puede encargarse de la comunicación con el usuario, mientras otros actúan en segundo plano, realizando la transmisión de un fichero, accediendo a recursos del sistema (cargar sonidos, leer ficheros ...), etc. De hecho todos los programas con interface gráfico (AWT o Swing) son multithread porque los eventos y las rutinas de dibujado de las ventanas corren en un hilo distinto al principal.

La Clase Thread

La forma más directa de crear un hilo es extender la clase Thread, del paquete java.lang, y redefinir el método run(), que contendrá el código a ejecutarse paralelamente a otros hilos del programa. Este método es invocado cuando se inicia el hilo, mediante el envío del mensaje start() de la clase Thread. El hilo inicia su ejecución con la llamada al método run y termina cuando termina éste.

La clase Thread implementa la interface Runnable, y provee, entre otros, los siguientes métodos:

- *currentThread():*devuelve el objeto thread que representa al hilo que se esta ejecutando.
- *yield ():* hace que el intérprete cambie el contexto entre el hilo actual y el siguiente hilo disponible
- *sleep(long):*provoca que el intérprete ponga al hilo en curso a dormir el número de milisegundos que se coloca como parámetro
- *start():*indica al interprete que comience a ejecutar el hilo.
- *run():* es el único método de la interfase Runnable. Es llamado por star()
- *stop():* provoca que el hilo se detenga de manera inmediata.
- *suspend():* toma el hilo y hace que detenga su ejecución sin destruirlo.
- *resume():* se utiliza para revivir un hilo suspendido.
- *setPriority(int):* asigna al hilo la prioridad indicada por el valor pasado como parámetro.
- *getPriority():* devuelve la prioridad del hilo de ejecución en curso.
- *setName(String)/getName():* asigna/ retorna nombre al/del hilo en ejecución.

Ejemplo 1:

```
public class Hilo extends Thread{
    public Hilo(String p_nbreHilo) {
        super(p_nbreHilo);
    }

    public void run(){
         /* Muestra información sobre el hilo actual */
        System.out.println("Soy el hilo --> " + Thread.currentThread().getName() );
    }
}

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

// Se crean tres hilos de la clase Hilo con los nombres Uno, Dos y Tres
            Hilo hilo1 = new Hilo("Uno");
            Hilo hilo2 = new Hilo("Dos");
            Hilo hilo3 = new Hilo("Tres");
```

```
        // Se inician los hilos. Al invocar al método start() se provoca que el hilo ejecute su método run()
              hilo1.start();
              hilo2.start();
              hilo3.start();
        }
}
```

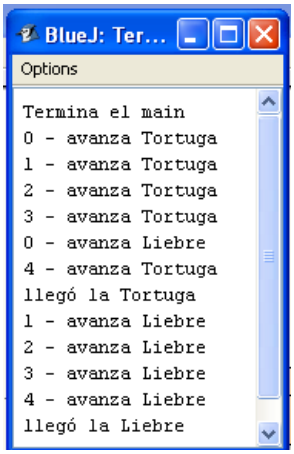
Resultado ejecución

Soy el hilo --> Uno
Soy el hilo --> Tres
Soy el hilo --> Dos

Ejemplo 2: Carrera de animales

```
public class Animal extends Thread{
    private String tipoAnimal;

    public Animal(String p_tipoAnimal){
            this.setTipoAnimal(p_tipoAnimal);
    }
    public void   setTipoAnimal(String p_tipoAnimal){this.tipoAnimal=p_tipoAnimal;}
    public String getTipoAnimal(){return this.tipoAnimal;}

    public void run(){
          for(int i = 0; i < 5 ; i++){
            System.out.println(i + " - avanza " + this.getTipoAnimal());
        }
       System.out.println("llegó la " + this.getTipoAnimal());
    }
}

public class Carrera{
       public static void main(String argv[]){
           Animal tortuga = new Animal("Tortuga");
           Animal liebre = new Animal("Liebre");

          //  La tortuga largará primero
           tortuga.start();
          // Luego la liebre
           liebre.start();
           System.out.println("Termina el main");
       }
}
```

Resultado ejecución



En los ejemplos se puede observar lo siguiente:

- La clase Thread está en el package java.lang por tanto no es necesario el import.

- En el ejemplo 1 el constructor recibe un parámetro que es la identificación del hilo (atributo *name* de la superclase Thread), con el que llama al constructor de la super().

```
public Thread(String str)
```

- En el ejemplo 2 el constructor recibe un parámetro que identifica al hilo (atributo de la subclase), que lo asigna a su atributo.

```
this.setTipoAnimal(p_tipoAnimal);
```

- En la salida del ejemplo 2 se observa en primer lugar el mensaje de finalización del main. **La ejecución de los threads es asíncrona**. Realiza la llamada al método start(), éste le devuelve el control y continua su ejecución, independiente de los otros threads.
- En la salida los mensajes de un thread y otro se van mezclando. La JVM asigna tiempos a cada thread.
- Si se ejecuta varias veces, se pueden obtener salidas en distinto orden

La Interface Runnable

Proporciona una alternativa a la utilización de la clase Thread, para los casos en los que no es posible extender la clase Thread. Esto ocurre cuando la clase que debe correr en un thread independiente necesita extender alguna otra clase. Dado que no existe herencia múltiple, no puede extender a la vez a la clase Thread. En este caso la clase debe implantar la interface Runnable, variando ligeramente la forma en que se crean e inician los nuevos threads.

El siguiente ejemplo es equivalente al anterior, pero utilizando la interface Runnable:

```
public class AnimalRn implements Runnable{
  private String tipoAnimal;

  public AnimalRn(String p_tipoAnimal){
        this.setTipoAnimal(p_tipoAnimal);
  }

  public void run(){
    for(int i = 0; i < 5 ; i++){
        System.out.println(i + " - avanza " + this.getTipoAnimal());
    }
    System.out.println("llegó la " + this.getTipoAnimal());
  }

  public void setTipoAnimal(String p_tipoAnimal){this.tipoAnimal=p_tipoAnimal;}
  public String getTipoAnimal(){return this.tipoAnimal;}
}


public class CarreraRn{
      public static void main(String argv[]){
         Thread tortuga = new Thread(new AnimalRn("Tortuga"));
         Thread liebre  = new Thread(new AnimalRn("Liebre"));

         tortuga.start();
         liebre.start();
         System.out.println("Termina el main");
      }
}
```

En esta implementación se observa que la clase AnimalRN implementa la interface Runnable (que expone el método *run*), en lugar de extender la clase Thread. En el main se crea el objeto Thread, utilizando otro constructor, que recibe un objeto runnable.

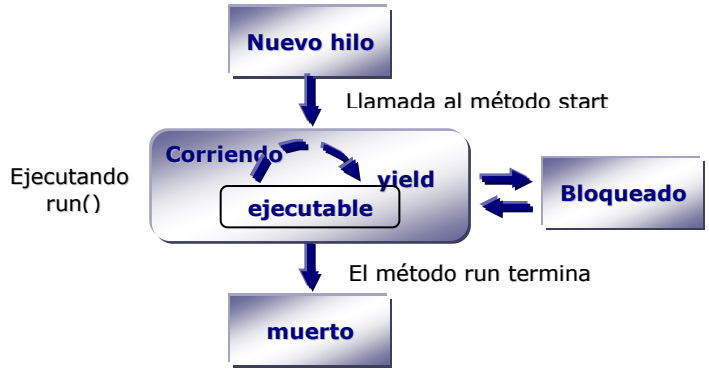
Ciclo de vida de un hilo

Para entender cómo trabaja un thread es necesario conocer los distintos estados en los cuales se puede encontrar:

1. **Nuevo:** Se ha reservado memoria para el thread y se han inicializado sus variables, pero no se ha puesto todavía en cola de ejecución. Dicho de otro modo: se ha creado un objeto de tipo Thread. En este estado pueden ocurrir dos cosas: que pase a cola de ejecución, mediante el método start(), o que se lo mate sin llegar a ejecutarse nunca, mediante el método stop().
2. **Ejecutable:** El thread está listo para ser ejecutado, se halla en cola de ejecución junto a los demás threads. Periódicamente la CPU cambia de thread de ejecución, tomando el thread que estaba ejecutando y poniéndolo en la cola de ejecución y escogiendo un nuevo thread de la cola de ejecución.

   Los threads llevan asociada una prioridad, cuando la CPU ha de elegir un thread de la cola de ejecución para ver cual ejecuta elige el de mayor prioridad, y de haber varios los va ejecutando secuencialmente.
3. **Corriendo:** El thread está siendo ejecutado en el procesador. Hay tres posibles causas por las que un thread que está corriendo pueda liberar el control del procesador:

   1- Se acaba su tiempo de ejecución y vuelve a la cola de procesos, pasando al estado de ejecutable.
   2- Se ejecuta el método sleep() o wait(), pasando a estado bloqueado
   3- Termina su cuerpo de ejecución, o se invoca a su método stop, por lo que pasa al estado muerto.
4. **Bloqueado:** En este estado el thread no está en la cola de ejecución, sino que está esperando a que ocurra un determinado evento o transcurra un determinado plazo de tiempo antes de poder acceder a la cola de ejecución.
5. **Muerto:** Como su nombre indica el thread deja de existir. Puede ocurrir bien porque ha terminado su cuerpo de ejecución o bien porque ha sido muerto mediante el método stop().



Tareas programadas

A menudo es necesario que un programa se interrumpa en determinados momentos para realizar ciertas tareas programadas. Para ello Java cuenta con las clases *Timer* y *TimerTask*

Todos los objetos *Timer* tienen un hilo asociado, de la clase TimerTask, que implementa Runnable.

Por ejemplo, se puede programar una tarea de la siguiente manera:

```
Timer t = new Timer(true);
t.scheduleAtFixedRate(new TareaProgramada(),0,1000);
```

Los valores 0 y 1000 pasados como argumento indican que el conteo de tiempo debe empezar inmediatamente y la tarea se haga cada mil milisegundos.

La clase TareaProgramada es una descendiente de TimerTask que debe definir el programador:

```
import java.utiI.TimerTask;
public class TareaProgramada extends TimerTask {
   public void run() {
    // aquí se define lo que se debe hacer
    }
}
```

Este ejemplo es muy simple, sin embargo, las clases TimerTask y Timer tienen un comportamiento más rico, que puede observarse a través de sus métodos públicos, en la documentación de Java.

Prioridades

Aunque un programa utilice varios hilos y aparentemente estos se ejecuten simultáneamente, el sistema ejecuta una única instrucción cada vez (esto es particularmente cierto en sistemas con una sola CPU), aunque las instrucciones se ejecutan concurrentemente (entremezclándose éstas). El mecanismo por el cual un sistema controla la ejecución concurrente de procesos se llama planificación (scheduling). Java soporta un mecanismo simple denominado planificación por prioridad fija (fixed priority scheduling). Esto significa que la planificación de los hilos se realiza en base a la prioridad relativa de un hilo frente a las prioridades de otros.

Java garantiza de cierta manera la gestión de los threads en cuanto al nivel de prioridad asignado a cada uno de ellos, de tal forma que cada hilo tiene una prioridad asociada. Si en un momento determinado un thread con prioridad mayor que el thread actual pasa a estado activo, éste pasa a ejecutarse y el actual será dormido. Dicho de otra forma, mientras el thread de mayor prioridad no esté dormido, es él el que contará con los recursos. Si se duerme, se pasa a los otros hilos la posibilidad de utilizar los recursos, pero si vuelve a pasar a activo el thread de mayor prioridad, éste recupera de nuevo los recursos. Por defecto los threads con idéntica prioridad se ejecutan con el sistema de Planificación Round-Robin[1], el cual define que si un thread comienza a ejecutarse, lo seguirá haciendo hasta que ocurra una de las siguientes circunstancias:

- Se llame a los métodos sleep() - wait() - yield() ó stop()
- Se espere una entrada/salida.
- Si se ejecuta un método sincronizado.
- Un hilo con una prioridad más alta esté en condiciones de ser ejecutado (runnable)

Tratamiento de las Prioridades

Cuando se crea un thread Java, hereda su prioridad desde el thread que lo ha creado. La prioridad por defecto de un thread está definida por la variable estática Thread.NORM_PRIORITY que tiene un valor por defecto de 5. Existen otras dos variables estáticas que son Thread.MIN_PRIORITY que está fijada a 1 (menor prioridad) y finalmente Thread.MAX_PRIORITY que tiene valor 10 (de mayor prioridad). Todas estas constantes se encuentran definidas en la clase Thread.

El método disponible para asignar la prioridad de un thread es setPriority(), el cual recibe como parámetro el valor con la prioridad para el thread, por tanto, un número entre uno y diez. Si se necesita cambiar la prioridad, normalmente se incrementa en uno o dos los valores de los estándares definidos anteriormente. Por ejemplo, especificar NORM_PRIORITY+1 es más que suficiente para reflejar un cambio de prioridad entre los hilos.

Si se desea saber qué prioridad tiene un thread en un momento determinado se dispone del método getPriority() que devuelve un número entero con el valor actual de prioridad.

Sólo cuando el thread se detiene, abandona o se convierte en "No Ejecutable" por alguna razón, empezará su ejecución un thread con prioridad inferior. El algoritmo de planificación de threads de Java también es preemptivo[2]: si en cualquier momento un thread con prioridad superior que todos los demás se vuelve "Ejecutable", el sistema elige el nuevo thread con prioridad más alta. Si dos o más hilos están listos para ejecutarse y tienen la misma prioridad, la máquina virtual va cediendo control de forma cíclica (round-robin). El hecho de que un hilo con una prioridad más alta interrumpa a otro se denomina 'planificación preventiva o apropiativa' (preemptive scheduling).

Cuando un hilo entra en ejecución y no cede voluntariamente el control para que puedan ejecutarse otros hilos, se dice que es un hilo egoísta (selfish thread). Algunos SO, como Windows, combaten estas actitudes con una estrategia de planificación por división de tiempos (time-slicing).

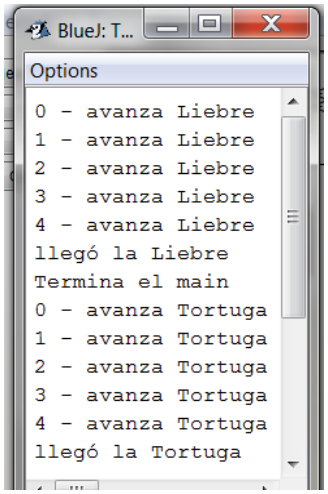
Ejemplo: Carrera de animales. Se agrega prioridad en el constructor

[1] método para seleccionar todos los elementos en un grupo de manera equitativa y en un orden racional

[2] preemptivo: capaz de suspender la ejecución de un proceso en cualquier momento en base a reglas de uso "justo" de los recursos

```
public class Animal extends Thread{
   private String tipoAnimal;
   public Animal(String p_tipoAnimal, int p_priority){
       this.setTipoAnimal(p_tipoAnimal);
       super.setPriority(p_priority);
   }
   public void run(){
       for(int i = 0; i < 5 ; i++){
          System.out.println(i + " - avanza " + this.getTipoAnimal());
       }
       System.out.println("llegó la " + this.getTipoAnimal());
  }
  public void setTipoAnimal(String p_tipoAnimal){this.tipoAnimal=p_tipoAnimal;}
  public String getTipoAnimal(){return this.tipoAnimal;}
}

public class Carrera{
   public static void main(String argv[]){
       Animal tortuga = new Animal("Tortuga",1);
       Animal liebre  = new Animal("Liebre",10);

      //  La tortuga largará primero
      tortuga.start();
      // Luego la liebre
      liebre.start();
      System.out.println("Termina el main");
   }
}
```

Resultado ejecución: se observa que el thread con prioridad superior (liebre) prevalece sobre los otros.



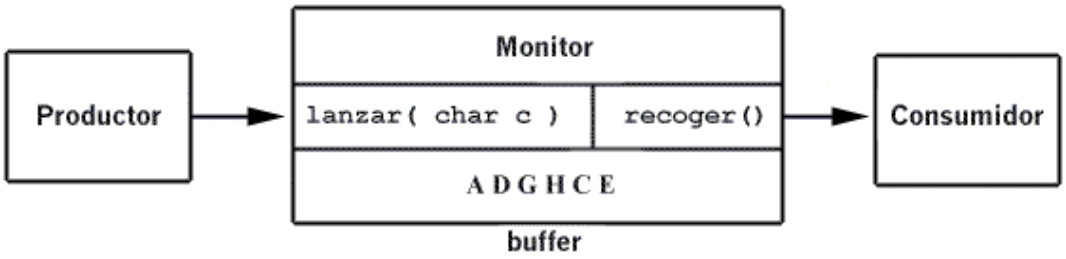
Threads Demonio (Daemons)

Se denomina "demonio" a un hilo que proporciona un servicio general, de fondo (background), mientras el programa se está ejecutando, pero no es parte de la esencia del programa. También se los conoce como *servicios*, porque se ejecutan, normalmente, con prioridad baja y proporcionan un servicio básico a un programa o programas cuando la actividad de la máquina es reducida. Un ejemplo de thread demonio que está ejecutándose continuamente es el recolector de basura (garbage collector). Este thread, proporcionado por la JVM, comprueba las variables de los programas a las que no se accede nunca y libera estos recursos, devolviéndolos al sistema. Un thread puede ser convertido a "demonio" pasandole el valor true al método setDaemon(). Si se pasa false a este método, el thread será devuelto por el sistema como un thread de usuario. No obstante, esto último debe realizarse antes de que se arranque el thread con start().Se puede saber si un hilo es un demonio llamando a *isDaemon*()

Comunicación entre hilos

Muchas veces los programas ejecutan varios hilos de forma asíncrona. Es decir, una vez que es iniciado, cada hilo vive de forma independiente de los otros, no existe ninguna relación entre ellos, ni tampoco ningún conflicto, dado que no comparten nada. Sin embargo, hay ocasiones que distintos hilos en un programa necesitan establecer alguna relación entre sí, o compartir objetos. Se necesita entonces algún mecanismo que permita la comunicación entre ellos, así como establecer 'reglas de juego' para acceder a recursos (objetos) compartidos.

Un ejemplo típico en que dos procesos se comunican es el caso en que un hilo produzca algún tipo de información que es procesada por otro hilo. Un thread produce una salida, que otro thread usa (consume), sea lo que sea esa salida. Al primer hilo se denomina en forma genérica *productor* y al segundo, *consumidor*. Existe también el concepto de *monitor*, que controlará el proceso de comunicación entre los threads. Funciona como una tubería: el productor inserta caracteres en un extremo y el consumidor los lee en el otro, con el monitor siendo la propia tubería.



Java proporciona un mecanismo elegante de comunicación entre procesos, a través de los métodos wait, notify y notifyAll. Estos métodos se implementan como métodos *final* en Object, de manera que todas las clases disponen de ellos.

- **wait:** le indica al hilo en curso que abandone el monitor y se vaya a dormir hasta que otro hilo entre en el mismo monitor y llame a notify.
- **notify:** despierta al primer hilo que realizó una llamada a wait sobre el mismo objeto.
- **notifyAll:** despierta todos los hilos que realizaron una llamada a wait sobre el mismo objeto. El hilo con mayor prioridad de los despertados es el primero en ejecutarse.

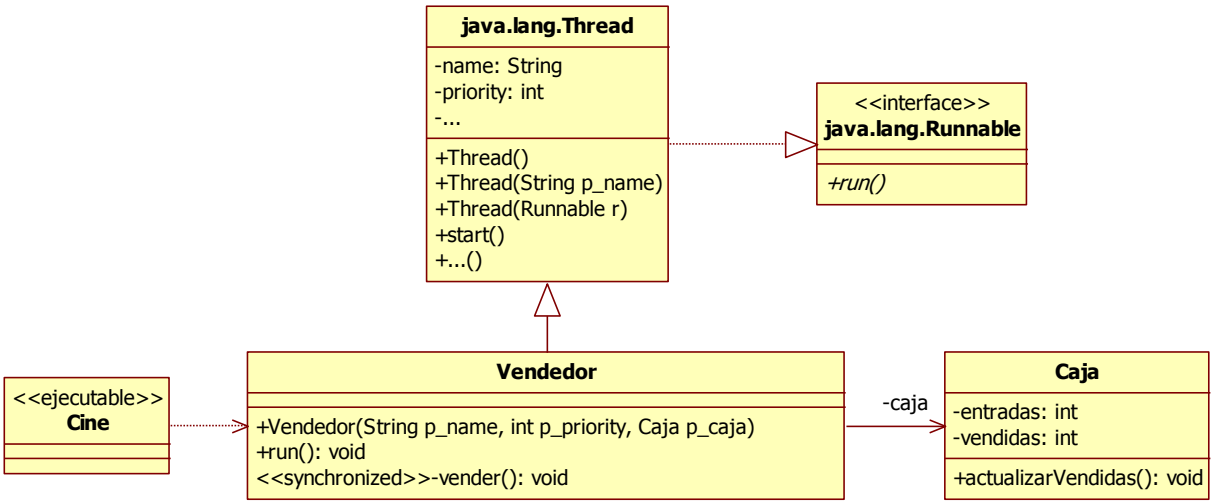
Cualquiera de los tres métodos sólo puede ser llamado desde dentro de un método synchronized.

Sincronización

La sincronización nace de la necesidad de evitar que dos o más hilos traten de acceder a los mismos recursos al mismo tiempo. Así, por ejemplo, si un hilo tratara de escribir en un fichero, y otro hilo estuviera al mismo tiempo tratando de borrar dicho fichero, se produciría una situación no deseada. Otra situación en la que hay que sincronizar hilos se produce cuando un hilo debe esperar a que estén preparados los datos que le debe suministrar el otro hilo. Para solucionar estos tipos de problemas es importante sincronizar los distintos hilos.

Las secciones de código de un programa que acceden a un mismo recurso (un mismo objeto de una clase, un fichero del disco, etc.) desde dos hilos distintos se denominan secciones críticas (critical sections). Para sincronizar dos o más threads, hay que utilizar el modificador synchronized en aquellos métodos del objeto-recurso con los que puedan producirse situaciones conflictivas. De esta forma, Java bloquea (asocia un bloqueo o lock) con el recurso sincronizado.

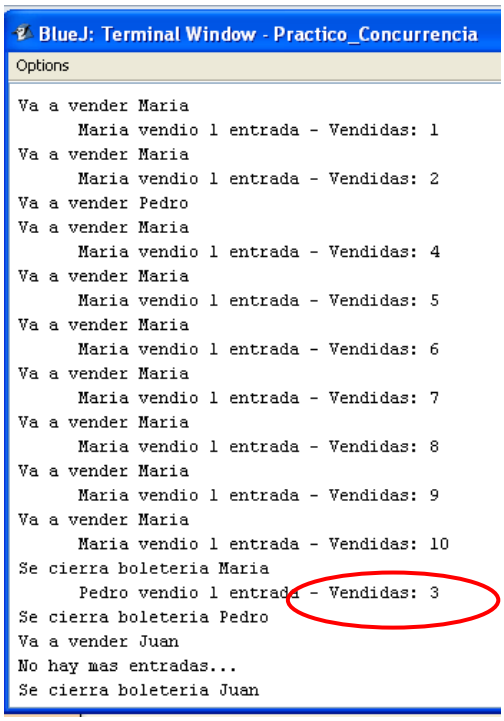
Ejemplo: En un cine se venden entradas en más de una boletería. Éstas deben estar sincronizadas, porque no se deben vender más entradas que la cantidad de asientos disponibles.

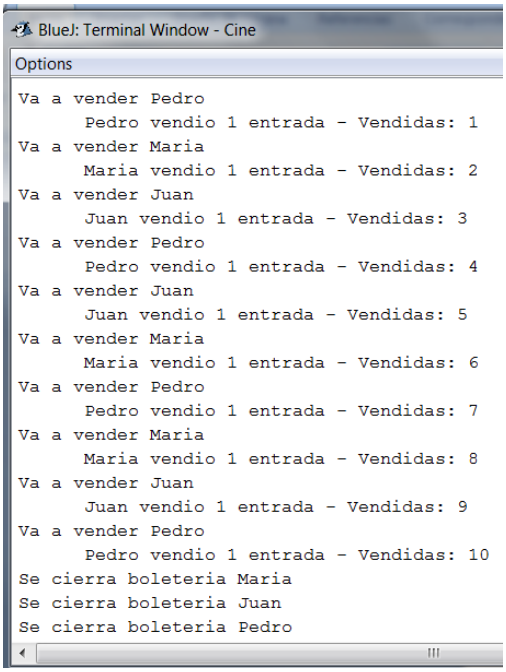



```
public class Caja{
    private int entradas;      // almacena total de entradas disponibles
    private int vendidas;      // almacena las entradas vendidas

    public Caja(int p_entradas){
       this.setEntradas(p_entradas);
       this.setVendidas(0);
    }
    private void setCaja(Caja p_caja){this.caja = p_caja;}
    public Caja getCaja(){return this.caja;}

 public void actualizarVendidas(){
        this.setVendidas(this.getVendidas() + 1);
    }

    private void setEntradas(int p_entradas){this.entradas = p_entradas;}
    public int getEntradas(){return this.entradas;}
    private void setVendidas(int p_vendidas){this.vendidas = p_vendidas;}
    public int getVendidas(){return this.vendidas;}
}
public class Vendedor extends Thread{
  private Caja caja;

  Vendedor(String p_name, int p_priority, Caja p_caja){
      super(p_name);
      super.setPriority(p_priority);
      this.setCaja(p_caja);
  }
  public void run(){
      while(this.getCaja().getVendidas() < this.getCaja().getEntradas()){
           this.vender();
      }
      System.out.println("Se cierra boleteria " +
                         Thread.currentThread().getName());
  }
  private synchronized void vender(){
     System.out.println("Va a vender "+Thread.currentThread().getName());
     if(this.getCaja().getVendidas() < this.getCaja().getEntradas()){
          this.getCaja().actualizarVendidas();
          System.out.println("      " + Thread.currentThread().getName() +
                             "vendio 1 entrada - " +
                             "Vendidas: " + this.getCaja().getVendidas());
     }else{
            System.out.println("No hay mas entradas... ");
          }
```

```
      // ********  las instrucciones siguientes son solo para que ejecute más lento
          try{
             Thread.sleep(10);
          }catch (InterruptedException ex) {
             ex.printStackTrace();
          }
    }
}

public class Cine{
    public static void main(String argv[]){
      // se crea una caja con la cantidad de entradas a vender
            Caja caja = new Caja(10);
      // se crean 3 vendedores que registran las ventas
           Vendedor v1 = new Vendedor("Juan",  5, caja);
           Vendedor v2 = new Vendedor("Pedro", 5, caja);
           Vendedor v3 = new Vendedor("Maria", 5, caja);
     // Comienzan a vender
           v1.start();
           v2.start();
           v3.start();
       }
    }
```

Resultado: en las imágenes se observa la diferencia entre ejecutar con el método sincronizado o no





Sincronización explícita de listas (arraylists) en java

Los **ArrayLists** no son sincronizados. Es decir que a pesar de que el método que utiliza una **ArrayList** esté sincronizado, otro hilo puede acceder a ella. Por esta razón, se debe sincronizar de forma explícita. Existen dos maneras de sincronización explícita de Listas en Java:

1. **CopyOnWriteArrayList:** no se abordará en esta asignatura
2. Utilizando el método **Collections.synchronizedList():** recibe como parámetro una lista **no sincronizada** (List), y retorna la misma lista, pero **sincronizada**.

List <Alumnos>listaAlumnos = new ArrayList();
List <Alumnos> listaSincronizadaAlumnos = Collections.synchronizedList(listaAlumnos);