3. PROGRAMACION CONCURRENTE

INTRODUCCION

Actualmente observamos que el paradigma orientado a objetos, solo podemos ejecutar un equipo a la vez como máximo en cambio con la introducción de las hebras concurrentes(programación concurrente) o procesos es posible que cada objeto se ejecute simultáneamente, esperando mensajes y respondiendo adecuadamente. Como siempre la principal razón para la investigación de la programación concurrente es que nos ofrece una manera diferente de conceptualizar la solución de un problema, una segunda razón es la de aprovechar el paralelismo del hardware subyacente para lograr una aceleración significativa.

Para entender mejor este detalle un buen ejemplo de un programa concurrente es el navegador Web de modem. Un ejemplo de concurrencia en un navegador Web se produce cuando e navegador empieza a presentar una página aunque puede estar aun descargando varios archivos de gráficos o de imágenes. La página que estamos presentando es un recurso compartido que deben gestionar cooperativamente las diversas hebras involucradas en la descarga de todos los aspectos de una página. Las diversas hebras no pueden escribir todas en la pantalla simultáneamente, especialmente si la imagen o grafico descargado provoca el cambio de tamaño del espacio asignado a la visualización de la imagen, afectando así la distribución del texto. Mientras hacemos todo esto hay varios botones que siguen activos sobre los que podemos hacer click particularmente el boton **stop** como una suerte de conclusión se observa en este paper que las hebras operan para llevar a cabo una tarea como la del ejemplo anterior así mismo se vera que los procesos deben tener acceso exclusivo aun recurso compartido como por ejemplo la visualización para evitar interferir unas con otras.

CONCEPTOS:

Ø *PROGRAMA SECUENCIAL:* Es aquel que especifica la ejecución de una secuencia de instrucciones que comprenden el programa.

Ø PROCESO: Es un programa en ejecución, tiene su propio estado independiente del estado de cualquier otro programa incluidos los del sistema operativo. Va acompañado de recursos como archivos, memoria, etc.

Ø *PROGRAMA CONCURRENTE:* Es un programa diseñado para tener 2 o mas contextos de ejecución decimos que este tipo de programa es multihenbrado, porque tiene mas de un contexto de ejecución.

Ø *PROGRAMA PARALELO*: Es un programa concurrente en el que hay mas de un contexto de ejecución o hebra activo simultáneamente; desde un punto de vista semántica no hay diferencia entre un programa paralelo y concurrente.

Ø *PROGRAMA DISTRIBUIDO*: Es un sistema diseñado para ejecutarse simultáneamente en una red de procesadores autónomos, que no comparten la memoria principal, con cada programa en su procesador aparte.

En un sistema operativo de multiprocesos el mismo programa lo pueden ejecutar múltiples procesos cada uno de ellos dando como resultado su propio estado o contexto de ejecución separado de los demás procesos. Ejm:

En la edición de documentos o arhivos de un programa en cada instancia del editor se llama de manera separada y se ejecuta en su propia ventana estos es claramente diferente de un programa multienhebrado en el que alguno de los datos residen simultáneamente en cada contexto de ejecución:

FUNDAMENTOS DE LA PROGRAMACIÓN CONCURRENTE

Concurrencia: Es un termino genérico utilizado para indicar un programa único en el que puede haber mas de un contexto de ejecución activo simultáneamente.

Comunicación entre procesos

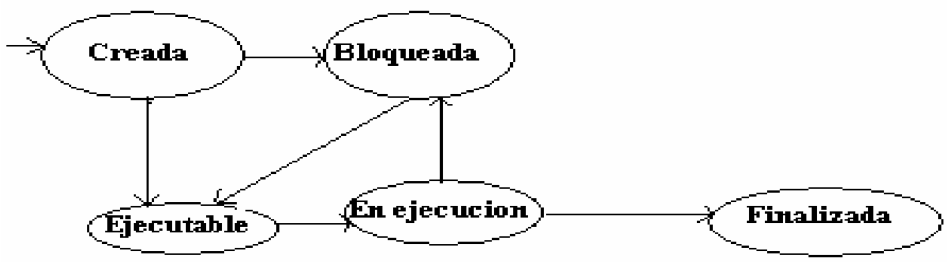
Introducción.-

Las aplicaciones informáticas pueden ser de muchos tipos desde procesadores de texto hasta procesadores de transacciones (Base de datos), simuladores de calculo intensivo, etc.

Las aplicaciones están formadas de uno o más programas. Los programas constan de código para la computadora donde se ejecutaran, el código es generalmente ejecutado en forma secuencial es así que normalmente un programa hilado o hebrado tiene el potencial de incrementar el rendimiento total de la aplicación en cuanto a productividad y tiempo de respuesta mediante ejecución de código asíncrono o paralelo.

Estados de una hebra (PROCESO):

1. **Creada**: La hebra se ha creado pero aun no esta lista para ejecutarse.
2. **Ejecutable o lista**: La hebra esta lista para ejecutarse pero espera a conseguir un procesador en que ejecutarse.
3. **En ejecución**: La hebra se esta ejecutando en un procesador.
4. **Bloqueado o en espera**: La hebra esta esperando tener acceso a una sección critica o ha dejado el procesador voluntariamente.
5. **Finalizada**: La hebra se ha parado y no se volverá a ejecutar.



Análisis de la comunicación entre procesos

Todos los programas concurrentes, implican interacción o comunicación entre hebras. Esto ocurre por las siguientes razones:

Ø Las hebras (incluso los procesos), compiten por un acceso exclusivo a los recursos compartidos, como los archivos físicos: archivos o datos.
Ø Las hebras se comunican para intercambiar datos.

En ambos casos es necesario que las hebras sincronicen su ejecución para evitar conflictos cuando adquieren los recursos, o para hacer contacto cuando intercambian datos. Una hebra puede comunicarse con otras mediante:

Ø Variables compartidas no locales: es el mecanismo principal utilizado por JAVA y también puede utilizarlo ADA.
Ø Paso de mensajes: es el mecanismo utilizado por ADA.
Ø Parámetro: lo utiliza ADA junto con el pasote mensajes.

Las hebras cooperan unas con otras para resolver un problema.

Exclusión mutua:

El método más sencillo de comunicación entre los procesos de un programa concurrente es el uso común de unas variables de datos. Esta forma tan sencilla de comunicación puede llevar, no obstante, a errores en el programa ya que el acceso concurrente puede hacer que la acción de un proceso interfiera en las acciones de otro de una forma no adecuada. Aunque nos vamos a fijar en variables de datos, todo lo que sigue sería válido con cualquier otro recurso del sistema que sólo pueda ser utilizado por un proceso a la vez. Por ejemplo una variable x compartida entre dos procesos A y B que pueden incrementar o decrementar la variable dependiendo de un determinado suceso. Éste situación se plantea, por ejemplo, en un problema típico de la programación concurrente conocido como el Problema de los Jardines.

En este problema se supone que se desea controlar el número de visitantes a unos jardines. La entrada y la salida a los jardines se pueden realizar por dos puntos que disponen de puertas giratorias. Se desea poder conocer en cualquier momento el número de visitantes a los jardines, por lo que se dispone de un computador con conexión en cada uno de los dos puntos de entrada que le informan cada vez que se produce una entrada o una salida. Asociamos el proceso P1 a un punto de entrada y el proceso P2 al otro punto de entrada. Ambos procesos se ejecutan de forma concurrente y utilizan una única variable x para llevar la cuenta del número de visitantes. El incremento o decremento de la variable se produce cada vez que un visitante entra o sale por una de las puertas. Así, la entrada de un visitante por una de las puertas hace que se ejecute la instrucción x:=x+1 mientras que la salida de un visitante hace que se ejecute la instrucción x:=x-1.

Si ambas instrucciones se realizaran como una única instrucción hardware, no se plantearía ningún problema y el programa podría funcionar siempre correctamente. Esto es así por que en un sistema con un único procesador sólo se puede realizar una instrucción cada vez y en un sistema multiprocesador se arbitran mecanismos que impiden que varios procesadores accedan a la vez a una misma posición de memoria. El resultado sería que el incremento o decremento de la variable se produciría de forma secuencial pero sin interferencia de un proceso en la acción del otro. Sin embargo, sí se produce interferencia de un proceso en el otro si la actualización de la variable se realiza mediante la ejecución de otras instrucciones más sencillas, como son las usuales de:

- Ø copiar el valor de x en un registro del procesador.
- Ø incrementar el valor del registro
- Ø almacenar el resultado en la dirección donde se guarda x

Aunque el proceso P1 y el P2 se suponen ejecutados en distintos procesadores (lo que no tiene porque ser cierto en la realidad) ambos usan la misma posición de memoria para guardar el valor de x.

Semáforos:

Se definieron originalmente en 1968 por dijkstra, fue presentado como un nuevo tipo de variable. Dijkstra una solución al problema de la exclusión mutua con la introducción del concepto de semáforo binario. Está técnica permite resolver la mayoría de los problemas de sincronización entre procesos y forma parte del diseño de muchos sistemas operativos y de lenguajes de programación concurrentes.

Un semáforo binario es un indicador (S) de condición que registra si un recurso está disponible o no. Un semáforo binario sólo puede tomar dos valores: 0 y 1. Si, para un semáforo binario, S = 1 entonces el recurso está disponible y la tarea lo puede utilizar; si S = 0 el recurso no está disponible y el proceso debe esperar.

Los semáforos se implementan con una cola de tareas o de condición a la cual se añaden los procesos que están en espera del recurso.

Sólo se permiten tres operaciones sobre un semáforo
- Inicializar
- Espera (wait)
- Señal (signal)

En algunos textos, se utilizan las notaciones P y V (*DOWN* y *UP*) para las operaciones de espera y señal respectivamente, ya que ésta fue la notación empleada originalmente por Dijkstra para referirse a las operaciones. Así pues, un semáforo binario se puede definir como un tipo de datos especial que sólo puede tomar los valores 0 y 1, con una cola de tareas asociada y con sólo tres operaciones para actuar sobre él.

Originalmente en el articulo de dijkstra, se utilizaron los nombres de P y V en lugar de *down* y *up,* pero ahora se utiliza los últimos que fueron presentados en ALGOL 60.

Interbloqueo e Injusticia

Se dice que una hebra esta en estado de ínter bloqueo, si esta esperando un evento que no se producirá nunca, el ínter bloqueo implica varias hebras, cada una de ellas a la espera de recursos existentes en otras. Un ínter bloqueo puede producirse, siempre que dos o más hebras compiten por recursos; para que existan ínter bloqueos son necesarias 4 condiciones:

- Las hebras deben reclamar derechos exclusivos a los recursos.
- Las hebras deben contener algún recurso mientras esperan otros; es decir, adquieren los recursos poco a poco, en vez de todos a la vez.
- No se pueden sacar recursos de hebras que están ala espera (no hay derecho preferente).
- Existe una cadena circular de hebras en la que cada hebra contiene uno o mas recursos necesarios para la siguiente hebra de la cadena

Sincronización

El uso de semáforos hace que se pueda programar fácilmente la sincronización entre dos tareas. En este caso las operaciones **espera** y **señal** no se utilizan dentro de un mismo proceso sino que se dan en dos procesos separados; el que ejecuta la operación de **espera** queda bloqueado hasta que el otro proceso ejecuta la operación de **señal**.

A veces se emplea la palabra **señal** para denominar un semáforo que se usa para sincronizar procesos. En este caso una **señal** tiene dos operaciones: *espera* y *señal* que utilizan para sincronizarse dos procesos distintos.

Monitores

Un monitor es un conjunto de procedimientos que proporciona el acceso con exclusión mutua a un recurso o conjunto de recursos compartidos por un grupo de procesos. Los procedimientos van encapsulados dentro de un módulo que tiene la propiedad especial de que sólo un proceso puede estar activo cada vez para ejecutar un procedimiento del monitor.

El monitor se puede ver como una valla alrededor del recurso (o recursos), de modo que los procesos que quieran utilizarlo deben entrar dentro de la valla, pero en la forma que impone el monitor. Muchos procesos pueden querer entrar en distintos instantes de tiempo, pero sólo se permite que entre un proceso cada vez, debiéndose esperar a que salga el que está dentro antes de que otro pueda entrar.

La ventaja para la exclusión mutua que presenta un monitor frente a los semáforos u otro mecanismo es que ésta está implícita: la única acción que debe realizar el programador del proceso que usa un recurso es invocar una entrada del monitor. Si el monitor se ha codificado correctamente no puede ser utilizado incorrectamente por un programa de aplicación que desee usar el recurso. Cosa que no ocurre con los semáforos, donde el programador debe proporcionar la correcta secuencia de operaciones *espera* y *señal* para no bloquear al sistema.

Los monitores no proporcionan por si mismos un mecanismo para la sincronización de tareas y por ello su construcción debe completarse permitiendo, por ejemplo, que se puedan usar **señales** para sincronizar los procesos. Así, para impedir que se pueda producir un bloqueo, un proceso que gane el acceso a un procedimiento del monitor y necesite esperar a una señal, se suspende y se coloca fuera del monitor para permitir que entre otro proceso.

Una variable que se utilice como mecanismo de sincronización en un monitor se conoce como **variable de condición**. A cada causa diferente por la que un proceso deba esperar se asocia una variable de condición. Sobre ellas sólo se puede actuar con dos procedimientos: **espera** y **señal**. En este caso, cuando un proceso ejecuta una operación de *espera* se suspende y se coloca en una cola asociada a dicha variable de condición. La diferencia con el semáforo radica en que ahora la ejecución de esta operación siempre suspende el proceso que la emite. La suspensión del proceso hace que se libere la posesión del monitor, lo que permite que entre otro proceso. Cuando un proceso ejecuta una operación de *señal* se libera un proceso suspendido en la cola de la variable de condición utilizada. Si no hay ningún proceso suspendido en la cola de la variable de condición invocada la operación *señal* no tiene ningún efecto.

Mensajes

Los mensajes proporcionan una solución al problema de la concurrencia de procesos que integra la sincronización y la comunicación entre ellos y resulta adecuado tanto para sistemas centralizados como distribuidos. Esto hace que se incluyan en prácticamente todos los sistemas operativos modernos y que en muchos de ellos se utilicen como base para todas las comunicaciones del sistema, tanto dentro del computador como en la comunicación entre computadores.

La comunicación mediante mensajes necesita siempre de un proceso emisor y de uno receptor así como de información que intercambiarse. Por ello, las operaciones básicas para comunicación mediante mensajes que proporciona todo sistema operativo son: **enviar** (*mensaje*) y **recibir** (*mensaje*). Las acciones de transmisión de información y de sincronización se ven como actividades inseparables.

La comunicación por mensajes requiere que se establezca un **enlace** entre el receptor y el emisor, la forma del cual puede variar grandemente de sistema a sistema. Aspectos importantes a tener en cuenta en los enlaces son: como y cuantos enlaces se pueden establecer entre los procesos, la capacidad de mensajes del enlace y tipo de los mensajes.

Su implementación varía dependiendo de tres aspectos:

- El modo de nombrar los procesos.
- El modelo de sincronización.
- Almacenamiento y estructura del mensaje.

ANALISIS DE CONCEPTOS MEDIANTE LOS EJEMPLOS CLASICOS DE LOS PROCESOS CONCURRENTES

El problema de la cena de los filósofos

En 1965 dijsktra planteo y resolvió un problema de sincronización llamado el problema de la cena de los filósofos, desde entonces todas las personas que idean cierta primitiva de sincronización, intentan demostrar lo maravilloso de la nueva primitiva al resolver este problema.

Se enuncia de la siguiente manera:

"5 filósofos se sientan en la mesa, cada uno tiene un plato de spaghetti, el spaghetti es tan escurridizo que un filósofo necesita dos tenedores para comerlo, entre cada dos platos hay un tenedor. La vida de un filosofo, consta de periodos alternados de comer y pensar, cuando un filosofo tiene hambre, intenta obtener un tenedor para su mano izquierda y otro para su mano derecha, alzando uno a la vez y en cualquier orden, si logra obtener los dos tenedores, come un rato y después deja los tenedores y continua pensando, la pregunta clave es: ¿puede usted escribir un programa, para cada filosofo que lleve a cabo lo que se supone debería y que nunca se detenga?, la solución obvia para este problema es que el procedimiento espere hasta que el tenedor especificado este disponible y se toman . Por desgracia esta solución es incorrecta. Supongamos que los 5 filósofos toman sus tenedores izquierdos en forma simultánea. Ninguno de ellos podría tomar su tenedor derecho con lo que ocurriría un bloqueo"[7].

Una situación como esta, en la que todos los programas se mantendrían por tiempo indefinido, pero sin hacer progresos se llama inanición

Decimos que una hebra esta *aplazada indefinidamente* si se retraza esperando un evento que puede no ocurrir nunca. La asignación de los recursos sobre una base de primero en entrar – primero en salir es una solución sencilla que elimina el desplazamiento indefinido.

Análogo al desplazamiento indefinido es el concepto de *injusticia* en este caso no se realiza ningún intento para asegurarse de que las hebras que tienen el mismo status hacen el mismo progreso en la adquisición de recursos, o sea, no toda acciones son igualmente probable.

Problema de los lectores y escritores:

Este problema modela el acceso a una base de datos. Imaginemos una enorme base de datos, como por ejemplo un sistema de reservaciones en una línea aérea con muchos procesos en competencia que intentan leer y escribir en ella.

Se puede aceptar que varios procesos lean la base de datos al mismo tiempo, pero si uno de los procesos esta escribiendo la base de datos, ninguno de los demás procesos debería tener acceso a esta, ni siquiera los lectores .La pregunta es: ¿Cómo programaría usted los lectores y escritores?.

Una posible solución es que el primer lector que obtiene el acceso, lleva a cabo un *down* En el *semáforo* dv. Los siguientes lectores solo incrementan un contador. Al salir los lectores estos decrementan el contador y el ultimo en salir hace un *up* en el semáforo, lo cual permite entrar a un escritor bloqueado si es que existe. Una hipótesis implícita en esta solución es que los lectores tienen prioridad entre los escritores, si surge un escritor mientras varios lectores se encuentran en la base de datos, el escritor debe esperar.

Pero si aparecen nuevos lectores, de forma que exista al menos un lector en la base de datos, el escritor deberá esperar hasta que no haya más lectores.

Este problema genero muchos puntos de vista tanto así que ***courtois etal*** también presento una solución que da prioridad a los escritores.

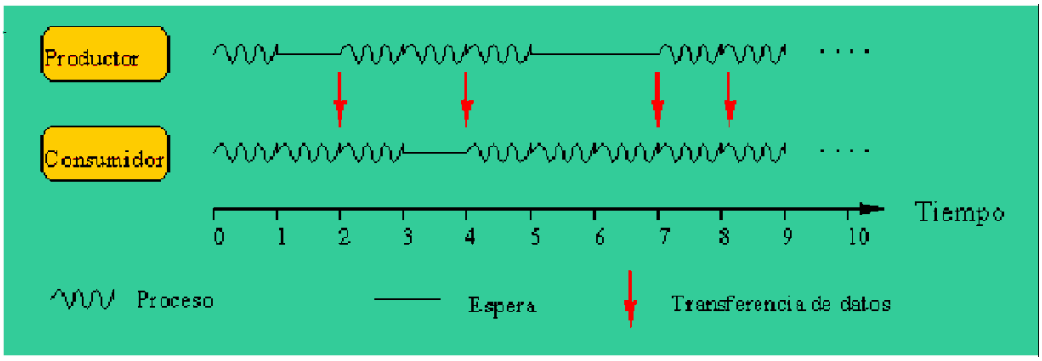
El problema del barbero dormilón:

Otro de los problemas clásicos de este paradigma ocurre en una peluquería, la peluquería tiene un barbero, una silla de peluquero y n sillas para que se sienten los clientes en espera, si no hay clientes presentes el barbero se sienta y se duerme. Cuando llega un cliente, este debe despertar al barbero dormilón. Si llegan mas clientes mientras el barbero corta el cabello de un cliente, ellos se sientan (si hay sillas desocupadas), o en todo caso, salen de la peluquería. El problema consiste en programar al barbero y los clientes sin entrar *en condiciones de competencia.* Una solución seria: cuando el barbero abre su negocio por la mañana ejecuta el procedimiento barber, lo que establece un bloqueo en el semáforo *"customers"*, hasta que alguien llega, después se va a dormir. Cuando llega el primer cliente el ejecuta *"customer"*. Si otro cliente llega poco tiempo después, el segundo no podrá hacer nada. Luego verifica entonces si el número de clientes que esperan es menor que el número de sillas, si esto no ocurre, sale sin su corte de pelo. Si existe una silla disponible, el cliente incrementa una variable contadora. Luego realiza un up en el semáforo *"customer"* con lo que despierta al barbero. En este momento tanto el cliente como el barbero están despiertos, luego cuando le toca su turno al cliente le cortan el pelo[7].

El problema del productor-consumidor.

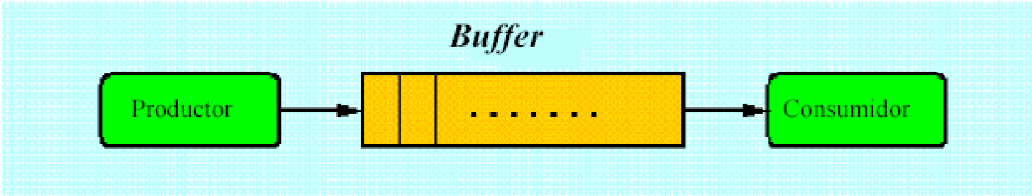
Estudiaremos problemas en la comunicación, supongamos dos tipos de procesos:
Productores y
Consumidores.

Productores: Procesos que crean elementos de datos mediante un procedimiento interno (Produce), estos datos deben ser enviados a los consumidores.
Consumidores: Procesos que reciben los elementos de datos creados por los productores, y que actúan en consecuencia mediante un procedimiento interno (Consume). Ejemplos: Impresoras, teclados, etc.

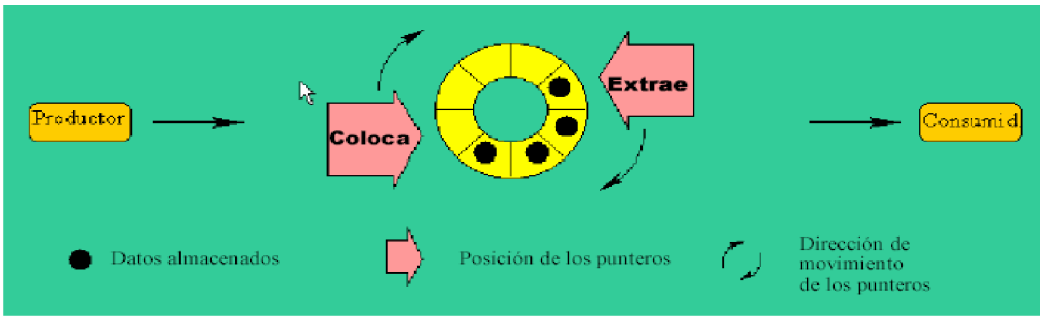


Problema: El productor envía un dato cada vez, y el consumidor consume un dato cada vez. Si uno de los dos procesos no está listo, el otro debe esperar.

Solución: Es necesario introducir un buffer en el proceso de transmisión de datos

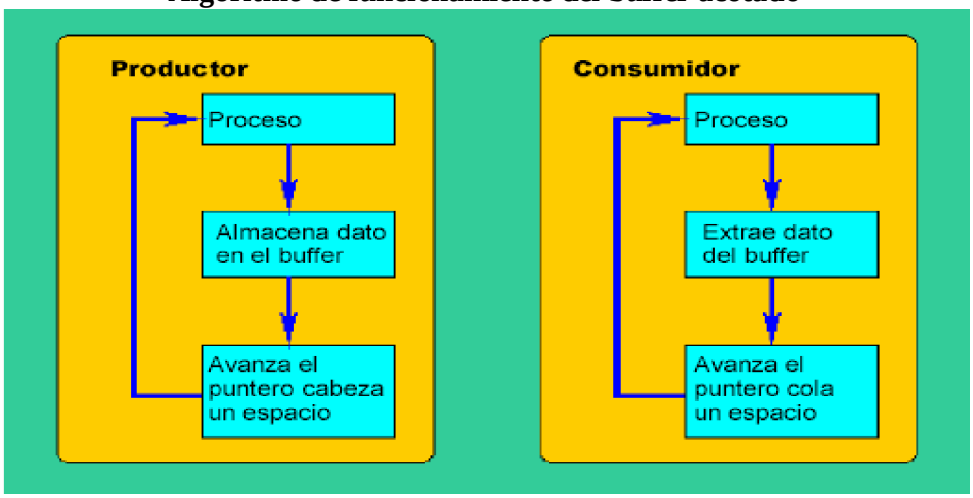


El buffer puede ser infinito. No obstante esto no es realista



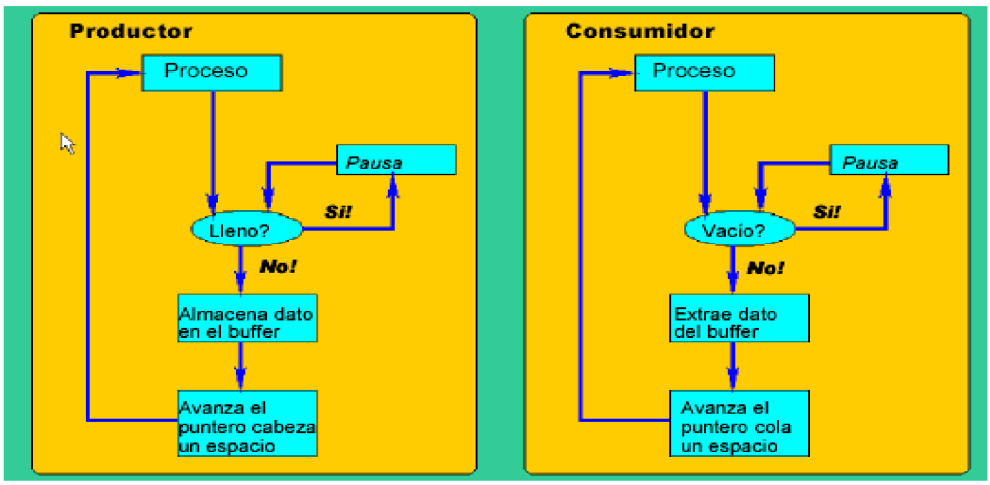
Alternativa: Buffer acotado en cola circular

Algoritmo de funcionamiento del buffer acotado



Problemas:
- Ø El productor puede enviar un dato a un buffer lleno
- Ø El consumidor puede extraer un dato de un buffer vacío

ES NECESARIO PREVENIR ESTAS SITUACIONES ANÓMALAS: Algoritmo de funcionamiento del buffer acotado



¿Cómo saber si el buffer está vacío o lleno?

Ø Una condición será un semáforo inicializado a un cierto valor.
Ø Necesitamos dos condiciones: lleno y vacío.
Ø Para añadir un dato al buffer, es necesario comprobar (wait) la condición lleno. SI se efectúa la operación con éxito se realiza un signal sobre la condición vacío.
Ø Para eliminar un dato del buffer, es necesario comprobar (wait) la condición vacío. SI se efectúa la operación con éxito se realiza un signal sobre la condición lleno.

CARACTERISTICAS DE LOS PROCESOS CONCURRENTES:

Interacción entre procesos: Los programas concurrentes implican interacción, entre los distintos procesos que los componen:

Ø Los procesos que compartan recursos y compiten por el acceso a los mismos.
Ø Los procesos que se comunican entre sí para intercambiar datos.

En ambas situaciones se necesitan que los procesos sincronicen su ejecución, para evitar conflictos o establecer contacto para el intercambio de datos .Esto se logra mediante variables compartidas o bien mediante el paso de mensajes.

Indeterminismo: las acciones que se especifican en un programa secuencial, tienen un orden total, pero en un programa concurrente el orden es parcial, ya que existe una incertidumbre sobre el orden exacto de concurrencia de ciertos sucesos. De esta forma si se ejecuta un programa concurrente varias veces, puede producir resultados diferentes partiendo de los mismos datos.

Gestión de recursos: los recursos compartidos necesitan una gestión especial. Un proceso que desea utilizar un recurso compartido debe solicitar dicho recurso, esperar a adquirirlo, utilizarlo y después liberarlo. Si el proceso solicita el recurso, pero no puede adquirirlo, en ese momento, es suspendido hasta que el recurso esta disponible.

La gestión de recursos compartidos es problemática y se debe realizar de tal forma que se eviten situaciones de retrazo indefinido o de bloqueo indefinido.

Comunicación: la comunicación entre procesos puede ser sincrona, cuando los procesos necesitan sincronizarse para intercambiar los datos, o asíncrona cuando un proceso que suministra los datos no necesita esperar a que el proceso receptor lo recoja, ya que los deja en un buffer de comunicación temporal.

Violación de la exclusión mutua: en ocasiones ciertas acciones que se realizan en un programa concurrente, no proporcionan los resultados deseados .Esto se debe a que existe una parte del programa donde se realizan dichas acciones que constituye una región critica, es decir, es una parte de un programa en la que se debe garantizar que si un proceso accede a la misma, ningún otro podrá acceder.

Bloqueo mutuo: un proceso se encuentra en estado de bloqueo mutuo si esta esperando por un suceso que no ocurrirá nunca. Se puede producir en la comunicación de procesos y en la gestión de recursos. Se basan en las siguientes condiciones:

- Ø Los procesos necesitan acceso exclusivo a los recursos.
- Ø Los procesos necesitan mantener ciertos recursos exclusivos y otros en espera.
- Ø Los recursos no se pueden obtener de los procesos que están a la espera.

PROGRAMACION EN JAVA

Hebras de java:

Una hebra puede estar en uno de estos cinco estados: creada, ejecutable, en ejecución, bloqueada o finalizada. Una hebra puede hacer transiciones de un estado a otro, ignorando en buena parte la transición del estado ejecutable al de la ejecución debido a que esto lo manipula el equipo virtual de java subyacente. En java como ocurre con todos los demás, una hebra es una clase por tanto la manera más sencilla de crear una hebra es crear una clase que herede de la clase thread:

```
public class mytrhead extends thread
{
public mythread()
{
……...
..……
}
}
```

Y hacer una operación *new* para crear una instancia de una thread: Thread thread = new mythread

();

Para hacer que una hebra creada sea ejecutable, llamamos sencillamente a su método *start:*

thread.start ( );

Después de algún coste adicional, la hebra recientemente ejecutable transfiere el control a su método *run* cada clase que amplia la clase thread debe proporcionar su propio método thread, pero no facilita un método *start,* basándose en el método start facilitado por la clase *thread,* normalmente el método *run* de una hebra contiene un bucle, ya que la salida de el método run finaliza la hebra. Por ejemplo en una animación grafica el método run movería repetidamente los objetos gráficos, repintaría la pantalla y después se dormiría (para ralentizar la animación), por tanto una animación grafica normal podría ser:

```
public void run{
        while ( true ){
                movedObjects( );
                repaint( );
                try { thread:sleep(50);
              } catch (InterruptedException exc){}
        }
}
```

Observemos que el método sleep, lanza potencialmente un InterruptedExeption, que debe captarse en una instrucción try- catch .

La llamada al método sleep mueve la hebra del estado en ejecución a un estado bloqueado, donde espera una interrupción del temporalizador de intervalo. La acción de dormir se realiza con frecuencia en las aplicaciones visuales, para evitar que la visualización se produzca demasiado rápido.

Un modo de conseguir un estado de finalizada, es hacer que la hebra salga de su método run, lo que después de un retrazo para la limpieza finalizaría la hebra, también se puede finalizar una hebra, llamando su método stock. Sin embargo, por razones complicadas, esto resulto ser problemático y en Java se desprecio el método stop de la clase thread. Una manera más sencilla de conseguir el mismo objetivo es considerar un booleano que el método run de la hebra utiliza para continuar el bucle por ejemplo:

```
Public void run( ){ While (continue ){
moveObjects( );
repaint( );
try{ thread.sleep(50);
}catch (InterruptedExecption exc){}
}
}
```

Para detener la hebra anterior, otra hebra llama a un método para establecer el valor de la variable de instancia *continue* en *false.* Esta variable no se considera compartida y podemos ignorar cualquier posible conexión de carrera de manera segura ya que como muchos provocara una interaccion extra del bucle.

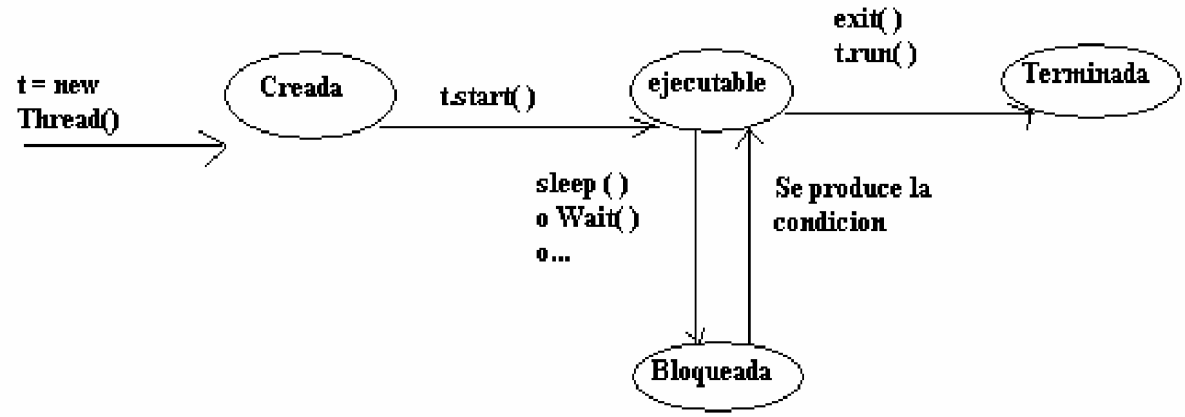
A veces no es conveniente hacer sus clases de la clase thread , por ejemplo, podemos querer que nuestra clase *applet* sea una hebra separada. En estos casos, una clase solo tiene que implementar la interfaz runnabl; es decir implementar un método *run,* el esquema es de este tipo de clase es :

```
public class myclass extends someClass implements Runnable{
…….
Public void run( ){…..}
}
```

hacemos una hebra con una instancia de myclass utilizando:

```
myclass obj = new myclass;
thread thread = new thread ( obj );
thread.start( );
```

Aquí vemos de nuevo una instancia en la que la utilización de interfaces de Java obvia la necesidad de herencia múltiple. La figura muestra los estados de una hebra de Java y las transiciones entre ellos:



SINCRONIZACION EN JAVA:

Básicamente, Java implementa el concepto de monitor de una manera bastante rigurosa asociando un bloqueo con cada objeto. Para implementar una variable compartida, creamos una clase de variable compartida e indicamos cada método (excepto el constructor) como sincronizado:

Public class sharedVariable…{
Public sharedVariable(…){….}

Public synchronized… method (…){…} Public synchronized… method (…){…}
…
}

para compartir una variable tenemos que crear una instancia de la clase y hacerla accesible para las hebras separadas . una manera de llevar estoa cabo seria pasar al objeto compartido, como parámetro del constructor de cada hebra. En el caso de una variable compartida de productor-consumidor, esto podría quedar así:

Shared Variable shared= new SharedVariable ( ); Thread producer = new Producer (shared);
Thread consumer = new (shared);

Donde damos por supuesto que tanto producerAndConsumer amplían switch.

PROGRAMACIÓN PARALELA

La **programación paralela** es una técnica de programación basada en la ejecución simultánea, bien sea en un mismo ordenador (con uno o varios procesadores) o en un cluster de ordenadores, en cuyo caso se denomina computación distribuida. Al contrario que en la programación concurrente, esta técnica enfatiza la verdadera simultaneidad en el tiempo de la ejecución de las tareas.

Los sistemas multiprocesador o multicomputador consiguen un aumento del rendimiento si se utilizan estas técnicas. En los sistemas monoprocesador el beneficio en rendimiento no es tan evidente, ya que la CPU es compartida por múltiples procesos en el tiempo, lo que se denomina multiplexación.

El mayor problema de la computación paralela radica en la complejidad de sincronizar unas tareas con otras, ya sea mediante secciones críticas, semáforos o paso de mensajes, para garantizar la exclusión mutua en las zonas del código en las que sea necesario.

Es el uso de varios procesadores trabajando juntos para resolver una tarea común, cada procesador trabaja una porción del problema pudiendo los procesos intercambiar datos a través de la memoria o por una red de interconexión.

Un programa paralelo es un programa concurrente en el que hay más de un contexto de ejecución, o hebra, activo simultáneamente. Cabe recalcar que desde un punto de vista semántica no hay diferencia entre un programa concurrente y uno paralelo[3].

Necesidad de la programación paralela

Ø Resolver problemas que no caben en una CPU.
Ø Resolver problemas que no se resuelven en un tiempo razonable.
Ø Se puede ejecutar :
O Problemas mayores.
O Problemas mas rápidamente.

Ø El rendimiento de los computadores secuenciales esta comenzando a saturarse, una posible solución seria usar varios procesadores, sistemas paralelos, con la tecnología VLSI(Very Large Scale Integration), el costo de los procesadores es menor.
Ø Decrementa la complejidad de un algoritmo al usar varios procesadores.

Aspectos en la programación paralela

Ø Diseño de computadores paralelos teniendo en cuenta la escalabilidad y comunicaciones.
Ø Diseño de algoritmos eficientes, no hay ganancia si los algoritmos no se diseñan adecuadamente.
Ø Métodos para evaluar los algoritmos paralelos: ¿Cuán rápido se puede resolver un problema usando una máquina paralela?,¿Con que eficiencia se usan esos procesadores?.
Ø En lenguajes para computadores paralelos deben ser flexibles para permitir una implementación eficiente y fácil de programar.
Ø Los programas paralelos deben ser portables y los compiladores paralelizantes.

Modelos de computadoras

Ø Modelo SISD
1. Simple instrucción, simple Data.
2. Consiste en un procesador que recibe un cierto flujo de instrucciones y de datos.
3. Un algoritmo para esta clase dice ser secuencial o serial.

Ø Modelo MISD
4. Multiple Instruction, simpledata.
5. Existencia de N procesadores con un simple flujo de datos ejecutando instrucciones diferentes en cada procesador.

Ø Modelo SIMD
6. Simple Instruction, múltiple Data.
7. Existen JN procesadores ejecutando la misma instrucción sobre los mismos o diferentes datos.
8. Existencia de datos compartidos.

Ø Modelo MIMD
9. Multiple Instruction, multiple data.
10. Existen N procesadores ejecutando instrucciones diferentes sobre los mismos o diferentes datos.
11. Existencia de datos compartidos.

CONCLUSIONES:

Ø Un programa paralelo es un programa concurrente en el que hay más de un contexto de ejecución, o hebra, activo simultáneamente.
Ø Las aplicaciones de la concurrencia se centra principalmente en los sistemas operativos.
Ø Se ha propuesto varias primitivas de la comunicación entre procesos(Semáforos, monitores, etc) pero todas son equivalentes en el sentido de que cada una se puede utilizar para implementar a las demás.
Ø La programación paralela ayuda a disminuir la complejidad computacional de los algoritmos.
Ø La programación paralela nos ayuda a resolver problemas más complejos.

ANEXOS COMPLEMENTARIO

Aplicaciones de las hebras

Algunos programas presentan una estructura que puede hacerles especialmente adecuados para entornos multihilo. Normalmente estos casos involucran operaciones que pueden ser solapadas. La utilización de múltiples hilos, puede conseguir un grado de paralelismo que incremente el rendimiento de un programa, e incluso hacer más fácil la escritura de su código.

Algunos ejemplos son:
Ø Utilización de los hilos para expresar *algoritmos inherentemente paralelos.* Se pueden obtener aumentos de rendimiento debido al hecho de que los hilos funcionan muy bien en sistemas multiprocesadores. Los hilos permiten expresar el paralelismo de alto nivel a través de un lenguaje de programación.

Ø Utilización de los hilos para *solapar operaciones de E/S lentas* con otras operaciones en un programa. Esto permite obtener un aumento del rendimiento, permitiendo bloquear a un simple hilo que espera a que se complete una operación de E/S, mientras otros hilos del proceso continúan su ejecución, evitando el bloqueo entero del proceso.
Ø Utilización de los hilos para *solapar llamadas RPC salientes*. Se obtiene una ganancia en el rendimiento permitiendo que un cliente pueda acceder a varios servidores al mismo tiempo en lugar de acceder a un servidor cada vez. Esto puede ser interesante para servidores especializados en los que los clientes pueden dividir una llamada RPC compleja en varias llamadas RPC concurrentes y más simples.
Ø Utilización de los *hilos en interfaces de usuario*. Se pueden obtener aumentos de rendimiento empleando un hilo para interactuar con un usuario, mientras se pasan las peticiones a otros hilos para su ejecución.
Ø Utilización de los *hilos en servidores*. Los servidores pueden utilizar las ventajas del multihilo, creando un hilo gestor diferente para cada petición entrante de un cliente.
Ø Utilización de los *hilos en procesos pipeline:* Se puede implementar cada etapa de una tubería o pipeline mediante un hilo separado dentro del mismo proceso.
Ø Utilización de los *hilos en el diseño de un kernel* multihilo de sistema operativo distribuido que distribuya diferentes tareas entre los hilos.
Ø Utilización de los *hilos para explotar la potencia de los multiprocesadores* de memori a compartida (sistemas fuertemente acoplados).
Ø Utilización de los *hilos como soporte de aplicaciones de tiempo real* acelerando los tiempos de respuesta para los eventos asíncronos a través de la gestión de señales.

Algunas de estas técnicas pueden ser implementadas usando múltiples procesos. Los hilos son sin embargo más interesantes como solución porque:

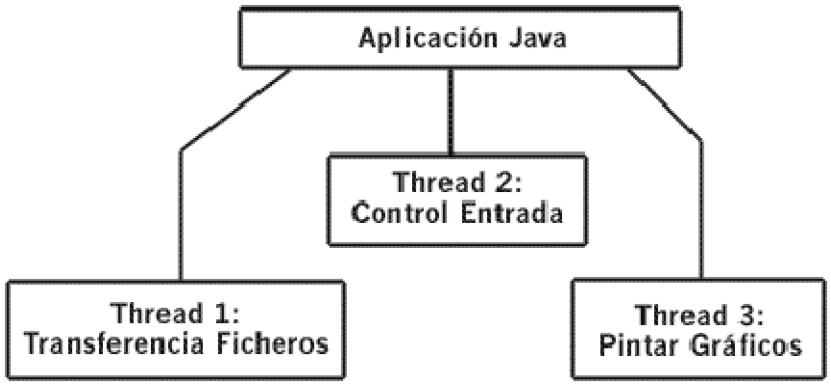
Ø Los procesos tienen un alto coste de creación.
Ø Los procesos requieren más memoria.
Ø Los procesos tienen un alto coste de sincronización.
Ø Compartir memoria entre procesos es más complicado, aunque compartir memoria entre hilos tiene sus problemas.

Los hilos se crearon para permitir combinar el paralelismo con la ejecución secuencial y el bloqueo de llamadas al sistema.

Las llamadas al sistema con bloqueo facilitan la programación, y el paralelismo obtenido mejora el rendimiento.

Hilos y Multihilo

Considerando el entorno *multithread* (multihilo), cada *thread* (hilo, flujo de control del programa) representa un proceso individual ejecutándose en un sistema. A veces se les llama *procesos ligeros* o *contextos de ejecución*. Típicamente, cada hilo controla un único aspecto dentro de un programa, como puede ser supervisar la entrada en un determinado periférico o controlar toda la entrada/salida del disco. Todos los hilos comparten los mismos recursos, al contrario que los *procesos*, en donde cada uno tiene su propia copia de código y datos (separados unos de otros). Gráficamente, los hilos (*threads*) se parecen en su funcionamiento a lo que muestra la figura siguiente:



Hay que distinguir *multihilo* (multithread) de *multiproceso*. El *multiproceso* se refiere a dos programas que se ejecutan "*aparentemente*" a la vez, bajo el control del Sistema Operativo. Los programas no necesitan tener relación unos con otros, simplemente el hecho de que el usuario desee que se ejecuten a la vez.

Multihilo se refiere a que dos o más tareas se ejecutan "*aparentemente*" a la vez, dentro de un mismo programa.

Se usa "*aparentemente*" en ambos casos, porque normalmente las plataformas tienen una sola CPU, con lo cual, los procesos se ejecutan en realidad "*concurrentemente*", sino que comparten la CPU. En plataformas con varias CPU, sí es posible que los procesos se ejecuten realmente a la vez.

Tanto en el multiproceso como en el multihilo (multitarea), el Sistema Operativo se encarga de que se genere la ilusión de que todo se ejecuta a la vez. Sin embargo, la multitarea puede producir programas que realicen más trabajo en la misma cantidad de tiempo que el multiproceso, debido a que la CPU está compartida entre tareas de un mismo proceso. Además, como el multiproceso está implementado a nivel de sistema operativo, el programador no puede intervenir en el planteamiento de su ejecución; mientras que en el caso del multihilo, como el programa debe ser diseñado expresamente para que pueda soportar esta característica, es imprescindible que el autor tenga que planificar adecuadamente la ejecución de cada hilo, o tarea.

Actualmente hay diferencias en la especificación del intérprete de Java, porque el intérprete de *Windows '95* conmuta los hilos de igual prioridad mediante un algoritmo circular (round-robin), mientras que el de *Solaris 2.X* deja que un hilo ocupe la CPU indefinidamente, lo que implica la inanición de los demás.

Programas de flujo único

Un programa de flujo único o mono-hilvanado (*single-thread*) utiliza un único flujo de control (*thread*) para controlar su ejecución. Muchos programas no necesitan la potencia o utilidad de múltiples flujos de control. Sin necesidad de especificar explícitamente que se quiere un único flujo de control, muchos de los applets y aplicaciones son de flujo único.

Por ejemplo, en la archiconocida aplicación estándar de saludo:

```
public class HolaMundo {
   static public void main( String args[] ) {
      System.out.println( "Hola Mundo!" );
      }
   }
```

Aquí, cuando se llama a *main()*, la aplicación imprime el mensaje y termina. Esto ocurre dentro de un único hilo de ejecución (*thread*).

Debido a que la mayor parte de los entornos operativos no solían ofrecer un soporte razonable para múltiples hilos de control, los lenguajes de programación tradicionales, tales como C++, no incorporaron mecanismos para describir de manera elegante situaciones de este tipo. La sincronización entre las múltiples partes de un programa se llevaba a cabo mediante un bucle de suceso único. Estos entornos son de tipo síncrono, gestionados por sucesos. Entornos tales como el de *Macintosh* de Apple, *Windows* de Microsoft y *X11/Motif* fueron diseñados en torno al modelo de bucle de suceso.

Programas de flujo múltiple

En la aplicación de saludo, no se ve el hilo de ejecución que corre el programa. Sin embargo, Java posibilita la creación y control de hilos de ejecución explícitamente. La utilización de hilos (*threads*) en Java, permite una enorme flexibilidad a los programadores a la hora de plantearse el desarrollo de aplicaciones. La simplicidad para crear, configurar y ejecutar hilos de ejecución, permite que se puedan implementar muy poderosas y portables aplicaciones/applets que no se puede con otros lenguajes de tercera generación. En un lenguaje orientado a Internet como es Java, esta herramienta es vital.

Si se ha utilizado un navegador con soporte Java, ya se habrá visto el uso de múltiples hilos en Java. Habrá observado que dos applets se pueden ejecutar al mismo tiempo, o que puede desplazar la página del navegador mientras el applet continúa ejecutándose. Esto no significa que el applet utilice múltiples hiloss, sino que el navegador es multihilo, multihilvanado o multithreaded.

Los navegadores utilizan diferentes hilos ejecutándose en paralelo para realizar varias tareas, "*aparentemente*" concurrentemente.

Por ejemplo, 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 las imágenes en un hilo de ejecución y soportando el desplazamiento de la página en otro hilo diferente.

Las aplicaciones (y applets) multihilo utilizan muchos contextos de ejecución para cumplir su trabajo. Hacen uso del hecho de que muchas tareas contienen subtareas distintas e independientes. Se puede utilizar un hilo de ejecución para cada subtarea.

Mientras que los programas de flujo único pueden realizar su tarea ejecutando las subtareas secuencialmente, un programa multihilo permite que cada thread comience y termine tan pronto como sea posible. Este comportamiento presenta una mejor respuesta a la entrada en tiempo real.

Vamos a modificar el programa de saludo creando tres hilos de ejecución individuales, que imprimen cada uno de ellos su propio mensaje de saludo, MultiHola.java:

```
// Definimos unos sencillos hilos. Se detendrán un rato
// antes de imprimir sus nombres y retardos

class TestTh extends Thread {
   private String nombre;
   private int retardo;

   // Constructor para almacenar nuestro nombre
   // y el retardo
   public TestTh( String s,int d ) {
      nombre = s;
      retardo = d;
      }

   // El metodo run() es similar al main(), pero para
   // threads. Cuando run() termina el thread muere
   public void run() {
      // Retasamos la ejecución el tiempo especificado
      try {
         sleep( retardo );
      } catch( InterruptedException e ) {
         ;
      }

      // Ahora imprimimos el nombre
      System.out.println( "Hola Mundo! "+nombre+" "+retardo );
      }
   }

public class MultiHola {
   public static void main( String args[] ) {
      TestTh t1,t2,t3;
```

```
    // Creamos los threads
    t1 = new TestTh( "Thread 1",(int)(Math.random()*2000) );
    t2 = new TestTh( "Thread 2",(int)(Math.random()*2000) );
    t3 = new TestTh( "Thread 3",(int)(Math.random()*2000) );

    // Arrancamos los threads
    t1.start();
    t2.start();
    t3.start();
  }
 }
```

Creación y Control de Hilos

Antes de entrar en más profundidades en los hilos de ejecución, se propone una referencia rápida de la clase Thread.

La clase Thread

Es la clase que encapsula todo el control necesario sobre los hilos de ejecución (*threads*). Hay que distinguir claramente un objeto *Thread* de un hilo de ejecución o *thread*. Esta distinción resulta complicada, aunque se puede simplificar si se considera al objeto *Thread* como el panel de control de un hilo de ejecución (*thread*). La clase **Thread** es la única forma de controlar el comportamiento de los hilos y para ello se sirve de los métodos que se exponen en las secciones siguientes.

Métodos de Clase

Estos son los métodos estáticos que deben llamarse de manera directa en la clase **Thread**.

currentThread()

Este método devuelve el objeto **thread** que representa al hilo de ejecución que se está ejecutando actualmente.

yield()

Este método hace que el intérprete cambie de contexto entre el hilo actual y el siguiente hilo ejecutable disponible. Es una manera de asegurar que nos hilos de menor prioridad no sufran inanición.

sleep(long)

El método *sleep()* provoca que el intérprete ponga al hilo en curso a dormir durante el número de milisegundos que se indiquen en el parámetro de invocación. Una vez transcurridos esos milisegundos, dicho hilo volverá a estar disponible para su ejecución.

Los relojes asociados a la mayor parte de los intérpretes de Java no serán capaces de obtener precisiones mayores de 10 milisegundos, por mucho que se permita indicar hasta nanosegundos en la llamada alternativa a este método.

Métodos de Instancia

Aquí no están recogidos todos los métodos de la clase **Thread**, sino solamente los más interesantes, porque los demás corresponden a áreas en donde el estándar de Java no está completo, y puede que se queden obsoletos en la próxima versión del JDK, por ello, si se desea completar la información que aquí se expone se ha de recurrir a la documentación del interfaz de programación de aplicación (API) del JDK.

start()

Este método indica al intérprete de Java que cree un contexto del hilo del sistema y comience a ejecutarlo. A continuación, el método *run()* de este hilo será invocado en el nuevo contexto del hilo. Hay que tener precaución de no llamar al método *start()* más de una vez sobre un hilo determinado.

run()

El método *run()* constituye el cuerpo de un hilo en ejecución. Este es el único método del interfaz **Runnable**. Es llamado por el método *start()* después de que el hilo apropiado del sistema se haya inicializado. Siempre que el método *run()* devuelva el control, el hilo actual se detendrá.

stop()

Este método provoca que el hilo se detenga de manera inmediata. A menudo constituye una manera brusca de detener un hilo, especialmente si este método se ejecuta sobre el hilo en curso. En tal caso, la línea inmediatamente posterior a la llamada al método *stop()* no llega a ejecutarse jamás, pues el contexto del hilo muere antes de que *stop()* devuelva el control. Una forma más elegante de detener un hilo es utilizar alguna variable que ocasione que el método *run()* termine de manera ordenada. En realidad, nunca se debería recurrir al uso de este método.

suspend()

El método *suspend()* es distinto de *stop()*. *suspend()* toma el hilo y provoca que se detenga su ejecución sin destruir el hilo de sistema subyacente, ni el estado del hilo anteriormente en ejecución. Si la ejecución de un hilo se suspende, puede llamarse a *resume()* sobre el mismo hilo para lograr que vuelva a ejecutarse de nuevo.

resume()

El método *resume()* se utiliza para revivir un hilo suspendido. No hay garantías de que el hilo comience a ejecutarse inmediatamente, ya que puede haber un hilo de mayor prioridad en ejecución actualmente, pero *resume()* ocasiona que el hilo vuelva a ser un candidato a ser ejecutado.

setPriority(int)

El método *setPriority()* asigna al hilo la prioridad indicada por el valor pasado como parámetro.

Hay bastantes constantes predefinidas para la prioridad, definidas en la clase **Thread**, tales como MIN_PRIORITY, NORM_PRIORITY y MAX_PRIORITY, que toman los valores 1, 5 y 10, respectivamente. Como guía aproximada de utilización, se puede establecer que la mayor parte de los procesos a nivel de usuario deberían tomar una prioridad en torno a NORM_PRIORITY. Las tareas en segundo plano, como una entrada/salida a red o el nuevo dibujo de la pantalla, deberían tener una prioridad cercana a MIN_PRIORITY. Con las tareas a las que se fije la máxima prioridad, en torno a MAX_PRIORITY, hay que ser especialmente cuidadosos, porque si no se hacen llamadas a *sleep()* o *yield(),* se puede provocar que el intérprete Java quede totalmente fuera de control.

getPriority()

Este método devuelve la prioridad del hilo de ejecución en curso, que es un valor comprendido entre uno y diez.

setName(String)

Este método permite identificar al hilo con un nombre menmónico. De esta manera se facilita la depuración de programas multihilo. El nombre mnemónico aparecerá en todas las líneas de trazado que se muestran cada vez que el intérprete Java imprime excepciones no capturadas.

getName()

Este método devuelve el valor actual, de tipo cadena, asignado como nombre al hilo en ejecución mediante *setName()*.

Creación de un Thread

Hay dos modos de conseguir hilos de ejecución (*threads*) en Java. Una es implementando el interfaz **Runnable**, la otra es extender la clase **Thread**.

La implementación del interfaz **Runnable** es la forma habitual de crear hilos. Los interfaces proporcionan al programador una forma de agrupar el trabajo de infraestructura de una clase. Se utilizan para diseñar los requerimientos comunes al conjunto de clases a implementar. El interfaz define el trabajo y la clase, o clases, que implementan el interfaz para realizar ese trabajo. Los diferentes grupos de clases que implementen el interfaz tendrán que seguir las mismas reglas de funcionamiento.

Hay unas cuantas diferencias entre interfaz y clase, que ya son conocidas y aquí solamente se resumen. Primero, un interfaz solamente puede contener métodos abstractos y/o variables estáticas y finales (constantes). Las clases, por otro lado, pueden implementar métodos y contener variables que no sean constantes. Segundo, un interfaz no puede implementar cualquier método. Una clase que implemente un interfaz debe implementar todos los métodos definidos en ese interfaz.

Un interfaz tiene la posibilidad de poder extenderse de otros interfaces y, al contrario que las clases, puede extenderse de múltiples interfaces.

Además, un interfaz no puede ser instanciado con el operador *new*; por ejemplo, la siguiente sentencia no está permitida:

```
Runnable a = new Runnable();   // No se permite
```

El primer método de crear un hilo de ejecución es simplemente extender la clase **Thread**:

```
class MiThread extends Thread {
   public void run() {
      . . .
      }
```

El ejemplo anterior crea una nueva clase **MiThread** que extiende la clase **Thread** y sobreescribe el método *Thread.run()* por su propia implementación. El método *run()* es donde se realizará todo el trabajo de la clase. Extendiendo la clase **Thread**, se pueden heredar los métodos y variables de la clase padre. En este caso, solamente se puede extender o derivar una vez de la clase padre. Esta limitación de Java puede ser superada a través de la implementación de *Runnable*:

```
public class MiThread implements Runnable {
   Thread t;
   public void run() {
      // Ejecución del thread una vez creado
      }
   }
```

En este caso necesitamos crear una instancia de *Thread* antes de que el sistema pueda ejecutar el proceso como un hilo. Además, el método abstracto *run()* está definido en el interfaz **Runnable** y tiene que ser implementado. La única diferencia entre los dos métodos es que este último es mucho más flexible. En el ejemplo anterior, todavía está la oportunidad de extender la clase **MiThread**, si fuese necesario. La mayoría de las clases creadas que necesiten ejecutarse como un hilo, implementarán el interfaz **Runnable**, ya que probablemente extenderán alguna de su funcionalidad a otras clases. No pensar que el interfaz **Runnable** está haciendo alguna cosa cuando la tarea se está ejecutando. Solamente contiene métodos abstractos, con lo cual es una clase para dar idea sobre el diseño de la clase **Thread**. De hecho, si se observan los fuentes de Java, se puede comprobar que solamente contiene un método abstracto:

```
package java.lang;
public interface Runnable {
   public abstract void run() ;
   }
```

Y esto es todo lo que hay sobre el interfaz **Runnable**. Como se ve, un interfaz sólo proporciona un diseño para las clases que vayan a ser implementadas.

En el caso de **Runnable**, fuerza a la definición del método *run()*, por lo tanto, la mayor parte del trabajo se hace en la clase **Thread**. Un vistazo un poco más profundo a la definición de la clase **Thread** da idea de lo que realmente está pasando:

```
public class Thread implements Runnable {
   ...
   public void run() {
      if( tarea != null )
         tarea.run() ;
         }
      }
   ...
   }
```

De este trocito de código se desprende que la clase **Thread** también implemente el interfaz **Runnable**. *tarea.run()* se asegura de que la clase con que trabaja (la clase que va a ejecutarse como un hilo) no sea nula y ejecuta el método *run()* de esa clase. Cuando esto suceda, el método *run()* de la clase hará que corra como un hilo.

A continuación se presenta un ejemplo, java1001.java, que implementa el interfaz **Runnable** para crear un programa multihilo.

```
class java1001 {
   static public void main( String args[] ) {
      // Se instancian dos nuevos objetos Thread
      Thread hiloA = new Thread( new MiHilo(),"hiloA" );
      Thread hiloB = new Thread( new MiHilo(),"hiloB" );

      // Se arrancan los dos hilos, para que comiencen su ejecución
      hiloA.start();
      hiloB.start();

      // Aquí se retrasa la ejecución un segundo y se captura la
      // posible excepción que genera el método, aunque no se hace
      // nada en el caso de que se produzca
      try {
         Thread.currentThread().sleep( 1000 );
      }catch( InterruptedException e ){}

      // Presenta información acerca del Thread o hilo principal
      // del programa
      System.out.println( Thread.currentThread() );

      // Se detiene la ejecución de los dos hilos
      hiloA.stop();
      hiloB.stop();
      }
   }
class NoHaceNada {
// Esta clase existe solamente para que sea heredada por la clase
// MiHilo, para evitar que esta clase sea capaz de heredar la clase
// Thread, y se pueda implementar el interfaz Runnable en su
// lugar
}
```

```
class MiHilo extends NoHaceNada implements Runnable {
   public void run() {
      // Presenta en pantalla información sobre este hilo en particular
      System.out.println( Thread.currentThread() );
      }
   }
```

Como se puede observar, el programa define una clase **MiHilo** que extiende a la clase **NoHaceNada** e implementa el interfaz **Runnable**. Se redefine el método *run()* en la clase **MiHilo** para presentar información sobre el hilo.

La única razón de extender la clase **NoHaceNada** es proporcionar un ejemplo de situación en que haya que extender alguna otra clase, además de implementar el interfaz.

En el ejemplo java1002.java muestra el mismo programa básicamente, pero en este caso extendiendo la clase **Thread**, en lugar de implementar el interfaz **Runnable** para crear el programa multihilo.

```
class java1002 {
   static public void main( String args[] ) {
      // Se instancian dos nuevos objetos Thread
      Thread hiloA = new Thread( new MiHilo(),"hiloA" );
      Thread hiloB = new Thread( new MiHilo(),"hiloB" );

      // Se arrancan los dos hilos, para que comiencen su ejecución
      hiloA.start();
      hiloB.start();

      // Aquí se retrasa la ejecución un segundo y se captura la
      // posible excepción que genera el método, aunque no se hace
      // nada en el caso de que se produzca
      try {
         Thread.currentThread().sleep( 1000 );
      }catch( InterruptedException e ){}
      // Presenta información acerca del Thread o hilo principal
      // del programa
      System.out.println( Thread.currentThread() );

      // Se detiene la ejecución de los dos hilos
      hiloA.stop();
      hiloB.stop();
      }
   }

class MiHilo extends Thread {
   public void run() {
      // Presenta en pantalla información sobre este hilo en particular
      System.out.println( Thread.currentThread() );
      }
   }
```

En ese caso, la nueva clase **MiHilo** extiende la clase **Thread** y no implementa el interfaz **Runnable** directamente (la clase **Thread** implementa el interfaz **Runnable**, por lo que indirectamente **MiHilo** también está implementando ese interfaz). El resto del programa es similar al anterior.

Y todavía se puede presentar un ejemplo más simple, utilizando un constructor de la clase **Thread** que no necesita parámetros, tal como se presenta en el ejemplo java1003.java. En los ejemplos anteriores, el constructor utilizado para **Thread** necesitaba dos parámetros, el primero un objeto de cualquier clase que implemente el interfaz **Runnable** y el segundo una cadena que indica el nombre del hilo (este nombre es independiente del nombre de la variable que referencia al objeto Thread).

```
class java1003 {
   static public void main( String args[] ) {
      // Se instancian dos nuevos objetos Thread
      Thread hiloA = new MiHilo();
      Thread hiloB = new MiHilo();

      // Se arrancan los dos hilos, para que comiencen su ejecución
      hiloA.start();
      hiloB.start();

      // Aquí se retrasa la ejecución un segundo y se captura la
      // posible excepción que genera el método, aunque no se hace
      // nada en el caso de que se produzca
      try {
         Thread.currentThread().sleep( 1000 );
      }catch( InterruptedException e ){}

      // Presenta información acerca del Thread o hilo principal
      // del programa
      System.out.println( Thread.currentThread() );

      // Se detiene la ejecución de los dos hilos
      hiloA.stop();
      hiloB.stop();
      }
   }

class MiHilo extends Thread {
   public void run() {
      // Presenta en pantalla información sobre este hilo en particular
      System.out.println( Thread.currentThread() );
      }
}
```

Las sentencias en este ejemplo para instanciar objetos Thread, son mucho menos complejas, siendo el programa, en esencia, el mismo de los ejemplos anteriores.

Arranque de un Thread

Las aplicaciones ejecutan *main()* tras arrancar. Esta es la razón de que *main()* sea el lugar natural para crear y arrancar otros hilos. La línea de código:

```
t1 = new TestTh( "Thread 1",(int)(Math.random()*2000) );
```

crea un nuevo hilo de ejecución. Los dos argumentos pasados representan el nombre del hilo y el tiempo que se desea que espere antes de imprimir el mensaje.

Al tener control directo sobre los hilos, hay que arrancarlos explícitamente. En el ejemplo con:

```
t1.start();
```

start(), en realidad es un método oculto en el hilo de ejecución que llama a *run()*.

Manipulación de un Thread

Si todo fue bien en la creación del hilo, t1 debería contener un thread válido, que controlaremos en el método *run()*. Una vez dentro de *run()*, se pueden comenzar las sentencias de ejecución como en otros programas. *run()* sirve como rutina *main()* para los hilos; cuando *run()* termina, también lo hace el hilo. Todo lo que se quiera que haga el hilo de ejecución ha de estar dentro de *run()*, por eso cuando se dice que un método es **Runnable**, es obligatorio escribir un método *run()*.

En este ejemplo, se intenta inmediatamente esperar durante una cantidad de tiempo aleatoria (pasada a través del constructor):

```
sleep( retardo );
```

El método *sleep()* simplemente le dice al hilo de ejecución que duerma durante los milisegundos especificados. Se debería utilizar *sleep()* cuando se pretenda retrasar la ejecución del hilo. *sleep()* no consume recursos del sistema mientras el hilo duerme. De esta forma otros hilos pueden seguir funcionando. Una vez hecho el retardo, se imprime el mensaje "Hola Mundo!" con el nombre del hilo y el retardo.

Suspensión de un Thread

Puede resultar útil suspender la ejecución de un hilo sin marcar un límite de tiempo. Si, por ejemplo, está construyendo un applet con un hilo de animación, seguramente se querrá permitir al usuario la opción de detener la animación hasta que quiera continuar.

No se trata de terminar la animación, sino desactivarla. Para este tipo de control de los hilos de ejecución se puede utilizar el método *suspend()*.

```
t1.suspend();
```

Este método no detiene la ejecución permanentemente. El hilo es suspendido indefinidamente y para volver a activarlo de nuevo se necesita realizar una invocación al método *resume()*:

```
t1.resume();
```

Parada de un Thread

El último elemento de control que se necesita sobre los hilos de ejecución es el método *stop()*. Se utiliza para terminar la ejecución de un hilo:

```
t1.stop();
```

Esta llamada no destruye el hilo, sino que detiene su ejecución. La ejecución no se puede reanudar ya con *t1.start()*. Cuando se desasignen las variables que se usan en el hilo, el objeto *Thread* (creado con new) quedará marcado para eliminarlo y el *garbage collector* se encargará de liberar la memoria que utilizaba.

En el ejemplo, no se necesita detener explícitamente el hilo de ejecución. Simplemente se le deja terminar. Los programas más complejos necesitarán un control sobre cada uno de los hilos que lancen, el método *stop()* puede utilizarse en esas situaciones.

Si se necesita, se puede comprobar si un hilo está vivo o no; considerando vivo un hilo que ha comenzado y no ha sido detenido.

```
t1.isAlive();
```

Este método devolverá *true* en caso de que el hilo t1 esté vivo, es decir, ya se haya llamado a su método *run()* y no haya sido parado con un *stop()* ni haya terminado el método *run()* en su ejecución.

En el ejemplo no hay problemas de realizar una parada incondicional, al estar todos los hilos vivos. Pero si a un hilo de ejecución, que puede no estar vivo, se le invoca su método *stop()*, se generará una excepción. En este caso, en los que el estado del hilo no puede conocerse de antemano es donde se requiere el uso del método *isAlive()*.

Grupos de Hilos

Todo hilo de ejecución en Java debe formar parte de un *grupo*. La clase **ThreadGroup** define e implementa la capacidad de un grupo de hilos.

Los grupos de hilos permiten que sea posible recoger varios hilos de ejecución en un solo objeto y manipularlo como un grupo, en vez de individualmente. Por ejemplo, se pueden regenerar los hilos de un grupo mediante una sola sentencia.

Cuando se crea un nuevo hilo, se coloca en un grupo, bien indicándolo explícitamente, o bien dejando que el sistema lo coloque en el grupo por defecto. Una vez creado el hilo y asignado a un grupo, ya no se podrá cambiar a otro grupo.

Si no se especifica un grupo en el constructor, el sistema coloca el hilo en el mismo grupo en que se encuentre el hilo de ejecución que lo haya creado, y si no se especifica en grupo para ninguno de los hilos, entonces todos serán miembros del grupo "*main*", que es creado por el sistema cuando arranca la aplicación Java.

En la ejecución de los ejemplos de esta sección, se ha podido observar la circunstancia anterior. Por ejemplo, el resultado en pantalla de uno de esos ejemplos es el que se reproduce a continuación:

```
% java java1002
Thread[hiloA,5,main]
Thread[hiloB,5,main]
Thread[main,5,main]
```

Como resultado de la ejecución de sentencias del tipo:

```
System.out.println( Thread.currentThread() );
```

Para presentar la información sobre el hilo de ejecución. Se puede observar que aparece el nombre del hilo, su prioridad y el nombre del grupo en que se encuentra englobado.

La clase **Thread** proporciona constructores en los que se puede especificar el grupo del hilo que se esta creando en el mismo momento de instanciarlo, y también métodos como *setThreadGroup()*, que permiten determinar el grupo en que se encuentra un hilo de ejecución.

Arrancar y Parar Threads

Ahora que ya se ha visto por encima como se arrancan, paran, manipulan y agrupan los hilos de ejecución, el ejemplo un poco más gráfico, java1004.java, implementa un contador.

El programa arranca un contador en 0 y lo incrementa, presentando su salida tanto en la pantalla gráfica como en la consola. Una primera ojeada al código puede dar la impresión de que el programa empezará a contar y presentará cada número, pero no es así. Una revisión más profunda del flujo de ejecución del applet, revelará su verdadera identidad.

En este caso, la clase **java1004** está forzada a implementar *Runnable* sobre la clase **Applet** que extiende. Como en todos los applets, el método *init()* es el primero que se ejecuta. En *init()*, la variable contador se inicializa a cero y se crea una nueva instancia de la clase **Thread**. Pasándole this al constructor de Thread, el nuevo hilo ya conocerá al objeto que va a correr. En este caso this es una referencia a java1004.

Después de que se haya creado el hilo, necesitamos arrancarlo. La llamada a *start()*, llamará a su vez al método *run()* de la clase, es decir, a *java1004.run()*. La llamada a *start()* retornará con éxito y el hilo comenzará a ejecutarse en ese instante. Observar que el método *run()* es un bucle infinito. Es infinito porque una vez que se sale de él, la ejecución del hilo se detiene. En este método se incrementará la variable contador, se duerme 10 milisegundos y envía una petición de refresco del nuevo valor al applet.

Es muy importante dormirse en algún lugar del hilo, porque sino, el hilo consumirá todo el tiempo de la CPU para su proceso y no permitirá que entren métodos de otros hilos a ejecutarse. Otra forma de detener la ejecución del hilo sería hacer una llamada al método *stop()*. En el contador, el hilo se detiene cuando se pulsa el ratón mientras el cursor se encuentre sobre el applet.

Dependiendo de la velocidad del ordenador, se presentarán los números consecutivos o no, porque el incremento de la variable contador es independiente del refresco en pantalla. El applet no se refresca a cada petición que se le hace, sino que el sistema operativo encolará las peticiones y las que sean sucesivas las convertirá en un único refresco. Así, mientras los refrescos se van encolando, la variable contador se estará todavía incrementando, pero no se visualiza en pantalla.

El uso y la conveniencia de utilización del método *stop()* es un poco dudoso y algo que quizá debería evitarse, porque puede haber objetos que dependan de la ejecución de varios hilos, y si se detiene uno de ellos, puede que el objeto en cuestión estuviese en un estado no demasiado consistente, y si se le mata el hilo de control puede que definitivamente ese objeto se dañe. Una solución alternativa es el uso de una variable de control que permita saber si el hilo se encuentra en ejecución o no, por ello, en el ejemplo se utiliza la variable miThread que controla cuando el hilo está en ejecución o parado.

La clase anidada **ProcesoRaton** es la que se encarga de implementar un objeto receptor de los eventos de ratón, para detectar cuando el usuario pulsa alguno de los botones sobre la zona de influencia del applet.

Suspender y Reanudar Threads

Una vez que se para un hilo de ejecución, ya no se puede rearrancar con el comando *start()*, debido a que *stop()* concluirá la ejecución del hilo. Por ello, en ver de parar el hilo, lo que se puede hacer es dormirlo, llamando al método *sleep()*. El hilo estará suspendido un cierto tiempo y luego reanudará su ejecución cuando el límite fijado se alcance. Pero esto no es útil cuando se necesite que el hilo reanude su ejecución ante la presencia de ciertos eventos. En estos casos, el método *suspend()* permite que cese la ejecución del hilo y el método *resume()* permite que un método suspendido reanude su ejecución.

En la versión modificada del ejemplo anterior, java1005.java, se modifica el applet para que utilice los métodos *suspend()* y *resume()*:

El uso de *suspend()* es crítico en ocasiones, sobre todo cuando el hilo que se va a suspender está utilizando recursos del sistema, porque en el momento de la suspensión los va a bloquear, y esos recursos seguirán bloqueados hasta que no se reanude la ejecución del hilo con *resume()*. Por ello, deben utilizarse métodos alternativos a estos, por ejemplo, implementando el uso de variables de control que vigiles periódicamente el estado en que se encuentra el hilo actual y obren el consecuencia.

```
public class java1005 extends Applet implements Runnable {

   ...
class ProcesoRaton extends MouseAdapter {
   boolean suspendido;
```

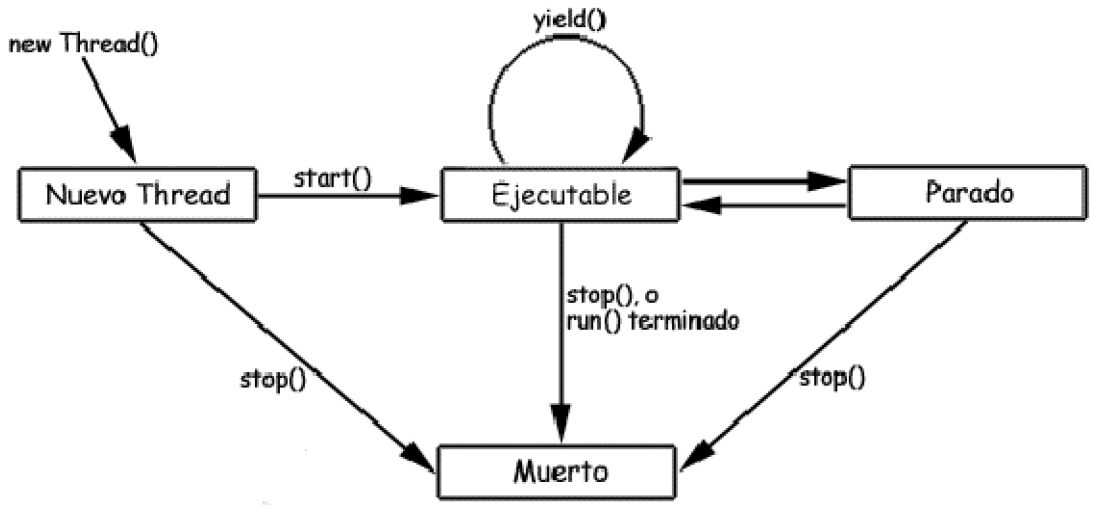
```
public void mousePressed( MouseEvent evt ) {
if( suspendido )
      t.resume();
   else
      t.suspend();
   suspendido = !suspendido;
   }
}
...
```

Para controlar el estado del applet, se ha modificado el funcionamiento del objeto **Listener** que recibe los eventos del ratón, en donde se ha introducido la variable suspendido. Diferenciar los distintos estados de ejecución del applet es importante porque algunos métodos pueden generar excepciones si se llaman desde un estado erróneo. Por ejemplo, si el applet ha sido arrancado y se detiene con *stop()*, si se intenta ejecutar el método *start()*, se generará una excepción *IllegalThreadStateException*.

Aquí podemos poner de nuevo en cuarentena la idoneidad del uso de estos métodos para el control del estado del hilo de ejecución, tanto por lo comentado del posible bloqueo de recursos vitales del sistema, como porque se puede generar un punto muerto en el sistema si el hilo de ejecución que va a intentar revivir el hilo suspendido necesita del recurso bloqueado. Por ello, es más seguro el uso de una variable de control como suspendido, de tal forma que sea ella quien controle el estado del hilo y utilizar el método *notify()* para indicar cuando el hilo vuelve a la vida.

Estados de un Hilo de Ejecución

Durante el ciclo de vida de un hilo, éste se puede encontrar en diferentes estados. La figura siguiente muestra estos estados y los métodos que provocan el paso de un estado a otro. Este diagrama no es una máquina de estados finita, pero es lo que más se aproxima al funcionamiento real de un hilo.



Nuevo Thread

La siguiente sentencia crea un nuevo hilo de ejecución pero no lo arranca, lo deja en el estado de *Nuevo Thread*:

```
Thread MiThread = new MiClaseThread();
Thread MiThread = new Thread( new UnaClaseThread,"hiloA" );
```

Cuando un hilo está en este estado, es simplemente un objeto *Thread* vacío. El sistema no ha destinado ningún recurso para él. Desde este estado solamente puede arrancarse llamando al método *start()*, o detenerse definitivamente, llamando al método *stop()*; la llamada a cualquier otro método carece de sentido y lo único que provocará será la generación de una excepción de tipo *IllegalThreadStateException*.

Ejecutable

Ahora obsérvense las dos líneas de código que se presentan a continuación:

```
Thread MiThread = new MiClaseThread();
MiThread.start();
```

La llamada al método *start()* creará los recursos del sistema necesarios para que el hilo puede ejecutarse, lo incorpora a la lista de procesos disponibles para ejecución del sistema y llama al método *run()* del hilo de ejecución. En este momento se encuentra en el estado *Ejecutable* del diagrama. Y este estado es *Ejecutable* y no *En Ejecución*, porque cuando el hilo está aquí no esta corriendo.

Muchos ordenadores tienen solamente un procesador lo que hace imposible que todos los hilos estén corriendo al mismo tiempo. Java implementa un tipo de *scheduling* o lista de procesos, que permite que el procesador sea compartido entre todos los procesos o hilos que se encuentran en la lista. Sin embargo, para el propósito que aquí se persigue, y en la mayoría de los casos, se puede considerar que este estado es realmente un estado *En Ejecución*, porque la impresión que produce ante el usuario es que todos los procesos se ejecutan al mismo tiempo.

Cuando el hilo se encuentra en este estado, todas las instrucciones de código que se encuentren dentro del bloque declarado para el método *run()*, se ejecutarán secuencialmente.

Parado

El hilo de ejecución entra en estado *Parado* cuando alguien llama al método *suspend()*, cuando se llama al método *sleep()*, cuando el hilo está bloqueado en un proceso de entrada/salida o cuando el hilo utiliza su método *wait()* para esperar a que se cumpla una determinada condición. Cuando ocurra cualquiera de las cuatro cosas anteriores, el hilo estará *Parado*.

Por ejemplo, en el trozo de código siguiente:

```
Thread MiThread = new MiClaseThread();
MiThread.start();
try {
   MiThread.sleep( 10000 );
} catch( InterruptedException e ) {
   ;
   }
```

la línea de código que llama al método *sleep()*:

```
   MiThread.sleep( 10000 );
```

hace que el hilo se duerma durante 10 segundos. Durante ese tiempo, incluso aunque el procesador estuviese totalmente libre, MiThread no correría. Después de esos 10 segundos. MiThread volvería a estar en estado *Ejecutable* y ahora sí que el procesador podría hacerle caso cuando se encuentre disponible.

Para cada una de los cuatro modos de entrada en estado *Parado*, hay una forma específica de volver a estado *Ejecutable*. Cada forma de recuperar ese estado es exclusiva; por ejemplo, si el hilo ha sido puesto a dormir, una vez transcurridos los milisegundos que se especifiquen, él solo se despierta y vuelve a estar en estado *Ejecutable*. Llamar al método *resume()* mientras esté el hilo durmiendo no serviría para nada.

Los métodos de recuperación del estado *Ejecutable*, en función de la forma de llegar al estado *Parado* del hilo, son los siguientes:

- Si un hilo está dormido, pasado el lapso de tiempo
- Si un hilo de ejecución está suspendido, después de una llamada a su método *resume()*
- Si un hilo está bloqueado en una entrada/salida, una vez que el comando de entrada/salida concluya su ejecución
- Si un hilo está esperando por una condición, cada vez que la variable que controla esa condición varíe debe llamarse al método *notify()* o *notifyAll()*

Muerto

Un hilo de ejecución se puede morir de dos formas: por causas naturales o porque lo maten (con *stop()*). Un hilo muere normalmente cuando concluye de forma habitual su método *run()*. Por ejemplo, en el siguiente trozo de código, el bucle while es un bucle finito -realiza la iteración 20 veces y termina-:

```
public void run() {
   int i=0;
   while( i < 20 ) {
      i++;
      System.out.println( "i = "+i );
      }
   }
```

Un hilo que contenga a este método *run()*, morirá naturalmente después de que se complete el bucle y *run()* concluya.

También se puede matar en cualquier momento un hilo, invocando a su método *stop()*. En el trozo de código siguiente:

```
Thread MiThread = new MiClaseThread();
MiThread.start();
try {
   MiThread.sleep( 10000 );
} catch( InterruptedException e ) {
   ;
   }
MiThread.stop();
```

se crea y arranca el hilo MiThread, se duerme durante 10 segundos y en el momento de despertarse, la llamada a su método *stop()*, lo mata.

El método *stop()* envía un objeto *ThreadDeath* al hilo de ejecución que quiere detener. Así, cuando un hilo es parado de este modo, muere asíncronamente. El hilo morirá en el momento en que reciba ese objeto *ThreadDeath*.

Los applets utilizarán el método *stop()* para matar a todos sus hilos cuando el navegador con soporte Java en el que se están ejecutando le indica al applet que se detengan, por ejemplo, cuando se minimiza la ventana del navegador o cuando se cambia de página.

El método *isAlive()*

El interfaz de programación de la clase **Thread** incluye el método *isAlive()*, que devuelve true si el hilo ha sido arrancado (con *start()*) y no ha sido detenido (con *stop()*). Por ello, si el método *isAlive()* devuelve false, sabemos que estamos ante un *Nuevo Thread* o ante un thread *Muerto*. Si devuelve true, se sabe que el hilo se encuentra en estado *Ejecutable* o *Parado*. No se puede diferenciar entre *Nuevo Thread* y *Muerto*, ni entre un hilo *Ejecutable* o *Parado*.

Scheduling

Java tiene un **Scheduler**, una lista de procesos, que monitoriza todos los hilos que se están ejecutando en todos los programas y decide cuales deben ejecutarse y cuales deben encontrarse preparados para su ejecución. Hay dos características de los hilos que el *scheduler* identifica en este proceso de decisión. Una, la más importante, es la prioridad del hilo de ejecución; la otra, es el indicador de *demonio*. La regla básica del *scheduler* es que si solamente hay *hilos demonio* ejecutándose, la *Máquina Virtual Java* (JVM) concluirá. Los nuevos hilos heredan la prioridad y el indicador de demonio de los hilos de ejecución que los han creado. El *scheduler* determina qué hilos deberán ejecutarse comprobando la prioridad de todos ellos, aquellos con prioridad más alta dispondrán del procesador antes de los que tienen prioridad más baja.

El *scheduler* puede seguir dos patrones, *preemptivo* y *no-preemptivo*. Los *schedulers preemtivos* proporcionan un segmento de tiempo a todos los hilos que están corriendo en el sistema. El *scheduler* decide cual será el siguiente hilo a ejecutarse y llama al método *resume()* para darle vida durante un período fijo de tiempo. Cuando el hilo ha estado en ejecución ese período de tiempo, se llama a *suspend()* y el siguiente hilo de ejecución en la lista de procesos será relanzado (*resume()*). Los *schedulers no-preemtivos* deciden que hilo debe correr y lo ejecutan hasta que concluye. El hilo tiene control total sobre el sistema mientras esté en ejecución. El método *yield()* es la forma en que un hilo fuerza al *scheduler* a comenzar la ejecución de otro hilo que esté esperando. Dependiendo del sistema en que esté corriendo Java, el *scheduler* será de un tipo u otro, preemtivo o no-preemptivo.

Prioridades

El *scheduler* determina el hilo que debe ejecutarse en función de la prioridad asignada a cada uno de ellos. El rango de prioridades oscila entre 1 y 10. La prioridad por defecto de un hilo de ejecución es NORM_PRIORITY, que tiene asignado un valor de 5. Hay otras dos variables estáticas disponibles, que son MIN_PRORITY, fijada a 1, y MAX_PRIORITY, que tiene un valor de 10. El método *getPriority()* puede utilizarse para conocer el valor actual de la prioridad de un hilo.

Hilos Demonio

Los hilos de ejecución *demonio* también se llaman *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.

Los hilos demonio son útiles cuando un hilo debe ejecutarse en segundo plano durante largos períodos de tiempo.Un ejemplo de *hilo demonio* que está ejecutándose continuamente es el recolector de basura (*garbage collector*).

Este hilo, proporcionado por la Máquina Virtual Java, comprueba las variables de los programas a las que no se accede nunca y libera estos recursos, devolviéndolos al sistema.

Un hilo puede fijar su indicador de demonio pasando un valor true al método *setDaemon()*. Si se pasa false a este método, el hilo de ejecución será devuelto por el sistema como un hilo de usuario. No obstante, esto último debe realizarse antes de que se arranque el hilo de ejecución (*start()*). Si se quiere saber si un hilo es un hilo demonio, se utilizará el método *isDaemon()*.

Diferencia entre hilos y *fork()*

fork() en Unix crea un proceso hijo que tiene su propia copia de datos y código del padre. Esto funciona correctamente si no hay problemas de cantidad de memoria de la máquina y se dispone de una CPU poderosa, y siempre que se mantenga el número de procesos hijos dentro de un límite manejable, porque se hace un uso intensivo de los recursos del sistema. Los applets Java no pueden *lanzar* ningún proceso en el cliente, porque eso sería una fuente de inseguridad y no está permitido. Las aplicaciones y los applets deben utilizar hilos de ejecución.

La *multitarea pre-emptiva* tiene sus problemas. Un hilo puede interrumpir a otro en cualquier momento, de ahí lo de *pre-emptive*. Fácilmente puede el lector imaginarse lo que pasaría si un hilo de ejecución está escribiendo en un array, mientras otro hilo lo interrumpe y comienza a escribir en el mismo array. Los lenguajes como C y C++ necesitan de las funciones *lock()* y *unlock()* para antes y después de leer o escribir datos. Java también funciona de este modo, pero oculta el bloqueo de datos bajo la sentencia synchronized:

```
synchronized int MiMetodo();
```

Otro área en que los hilos son muy útiles es en los interfaces de usuario. Permiten incrementar la respuesta del ordenador ante el usuario cuando se encuentra realizando complicados cálculos y no puede atender a la entrada de usuario. Estos cálculos se pueden realizar en segundo plano, o realizar varios en primer plano (música y animaciones) sin que se dé apariencia de pérdida de rendimiento.

Ejemplo de animación

Este es un ejemplo de un applet, java1006.java, que crea un hilo de animación que nos presenta el globo terráqueo en rotación. Aquí se puede ver que el applet crea un hilo de ejecución de sí mismo, **concurrencia**. Además, *animacion.start()* llama al *start()* del hilo, no del applet, que automáticamente llamará a *run()*:

```
import java.awt.*;
import java.applet.Applet;
public class java1006 extends Applet implements Runnable {
   Image imagenes[];
   MediaTracker tracker;
   int indice = 0;
   Thread animacion;

   int maxAncho,maxAlto;
   Image offScrImage; // Componente off-screen para doble buffering
   Graphics offScrGC;

   // Nos indicará si ya se puede pintar
   boolean cargado = false;

   // Inicializamos el applet, establecemos su tamaño y
   // cargamos las imágenes
   public void init() {
      // Establecemos el supervisor de imágenes
      tracker = new MediaTracker( this );
      // Fijamos el tamaño del applet
      maxAncho = 100;
      maxAlto = 100;
      imagenes = new Image[33];
```

```
    // Establecemos el doble buffer y dimensionamos el applet
    try {
        offScrImage = createImage( maxAncho,maxAlto );
        offScrGC = offScrImage.getGraphics();
        offScrGC.setColor( Color.lightGray );
        offScrGC.fillRect( 0,0,maxAncho,maxAlto );
        resize( maxAncho,maxAlto );
    } catch( Exception e ) {
        e.printStackTrace();
        }

    // Cargamos las imágenes en un array
    for( int i=0; i < 33; i++ )
        {
        String fichero =
            new String( "Tierra"+String.valueOf(i+1)+".gif" );
        imagenes[i] = getImage( getDocumentBase(),fichero );
        // Registramos las imágenes con el tracker
        tracker.addImage( imagenes[i],i );
        }

    try {
        // Utilizamos el tracker para comprobar que todas las
        // imágenes están cargadas
        tracker.waitForAll();
    } catch( InterruptedException e ) {
        ;
        }
    cargado = true;
    }

// Pintamos el fotograma que corresponda
public void paint( Graphics g ) {
    if( cargado )
        g.drawImage( offScrImage,0,0,this );
    }

// Arrancamos y establecemos la primera imagen
public void start() {
    if( tracker.checkID( indice ) )
        offScrGC.drawImage( imagenes[indice],0,0,this );
    animacion = new Thread( this );
    animacion.start();
    }

// Aquí hacemos el trabajo de animación
// Muestra una imagen, para, muestra la siguiente...
public void run() {
    // Obtiene el identificador del thread
    Thread thActual = Thread.currentThread();
```

```
    // Nos aseguramos de que se ejecuta cuando estamos en un
    // thread y además es el actual
    while( animacion != null  &&  animacion == thActual )
       {
       if( tracker.checkID( indice ) )
          {
          // Obtenemos la siguiente imagen
          offScrGC.drawImage( imagenes[indice],0,0,this );
          indice++;
          // Volvemos al principio y seguimos, para el bucle
          if( indice >= imagenes.length )
             indice = 0;
          }

       // Ralentizamos la animación para que parezca normal
       try {
          animacion.sleep( 200 );
       } catch( InterruptedException e ) {
          ;
          }
       // Pintamos el siguiente fotograma
       repaint();
       }
    }
}
```

En el ejemplo se pueden observar más cosas. La variable thActual es propia de cada hilo que se lance, y la variable animación la estarán viendo todos los hilos. No hay duplicidad de procesos, sino que todos comparten las mismas variables; cada hilo de ejecución, sin embargo, tiene su pila local de variables, que no comparte con nadie y que son las que están declaradas dentro de las llaves del método *run()*.

La excepción *InterruptedExcepcion* salta en el caso en que se haya tenido al hilo parado más tiempo del debido.

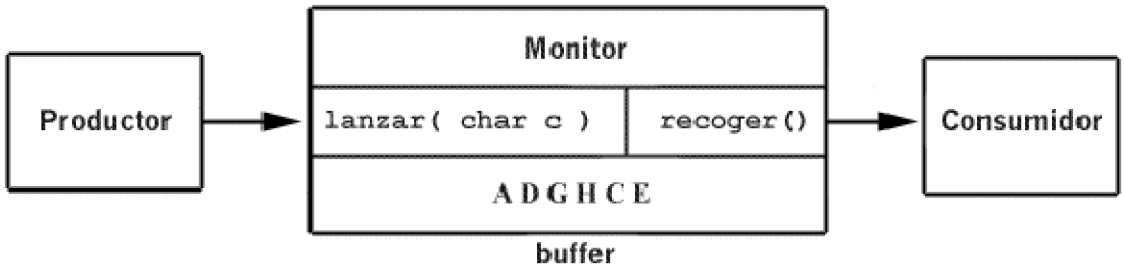
Es imprescindible recoger esta excepción cuando se están implementando hilos de ejecución, tanto es así, que en el caso de no recogerla, el compilador generará un error.

Comunicación entre Hilos

Otra clave para el éxito y la ventaja de la utilización de múltiples hilos de ejecución en una aplicación, o aplicación *multithreaded*, es que pueden comunicarse entre sí. Se pueden diseñar hilos para utilizar objetos comunes, que cada hilo puede manipular independientemente de los otros hilos de ejecución.

El ejemplo clásico de comunicación de hilos de ejecución es un modelo productor/consumidor. Un hilo produce una salida, que otro hilo usa (consume), sea lo que sea esa salida.

Entonces se crea un *productor*, que será un hilo que irá sacando caracteres por su salida; y se crea también un *consumidor* que irá recogiendo los caracteres que vaya sacando el productor y un *monitor* que controlará el proceso de sincronización entre los hilos de ejecución. Funcionará como una tubería, insertando el productor caracteres en un extremo y leyéndolos el consumidor en el otro, con el monitor siendo la propia tubería.



Productor

El productor extenderá la clase **Thread**, y su código es el siguiente:

```
class Productor extends Thread {
   private Tuberia tuberia;
   private String alfabeto = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";

   public Productor( Tuberia t ) {
      // Mantiene una copia propia del objeto compartido
      tuberia = t;
      }

   public void run() {
      char c;

      // Mete 10 letras en la tubería
      for( int i=0; i < 10; i++ )
         {
         c = alfabeto.charAt( (int)(Math.random()*26 ) );
         tuberia.lanzar( c );
         // Imprime un registro con lo añadido
         System.out.println( "Lanzado "+c+" a la tuberia." );
         // Espera un poco antes de añadir más letras
         try {
            sleep( (int)(Math.random() * 100 ) );
         } catch( InterruptedException e ) {;}
         }
      }
   }
```

Notar que se crea una instancia de la clase **Tuberia**, y que se utiliza el método *tuberia.lanzar()* para que se vaya construyendo la tubería, en principio de 10 caracteres.

Consumidor

Ahora se reproduce el código del consumidor, que también extenderá la clase **Thread**:

```
class Consumidor extends Thread {
   private Tuberia tuberia;

   public Consumidor( Tuberia t ) {
      // Mantiene una copia propia del objeto compartido
      tuberia = t;
      }

   public void run() {
      char c;

      // Consume 10 letras de la tubería
      for( int i=0; i < 10; i++ )
         {
         c = tuberia.recoger();
         // Imprime las letras retiradas
         System.out.println( "Recogido el caracter "+c );
         // Espera un poco antes de coger más letras
         try {
            sleep( (int)(Math.random() * 2000 ) );
         } catch( InterruptedException e ) {;}
         }
      }
   }
```

En este caso, como en el del productor, se cuenta con un método en la clase **Tuberia**, *tuberia.recoger()*, para manejar la información.

Monitor

Una vez vistos el productor de la información y el consumidor, solamente queda por ver qué es lo que hace la clase **Tuberia**.

Lo que realiza la clase **Tuberia**, es una función de supervisión de las transacciones entre los dos hilos de ejecución, el productor y el consumidor. Los monitores, en general, son piezas muy importantes de las aplicaciones multihilo, porque mantienen el flujo de comunicación entre los hilos.

```
class Tuberia {
   private char buffer[] = new char[6];
   private int siguiente = 0;
   // Flags para saber el estado del buffer
   private boolean estaLlena = false;
   private boolean estaVacia = true;
```

```
  // Método para retirar letras del buffer
  public synchronized char recoger() {
     // No se puede consumir si el buffer está vacío
     while( estaVacia == true )
        {
        try {
           wait(); // Se sale cuando estaVacia cambia a false
        } catch( InterruptedException e ) {
           ;
           }
        }
     // Decrementa la cuenta, ya que va a consumir una letra
     siguiente--;
     // Comprueba si se retiró la última letra
     if( siguiente == 0 )
        estaVacia = true;
     // El buffer no puede estar lleno, porque acabamos
     // de consumir
     estaLlena = false;
     notify();

     // Devuelve la letra al thread consumidor
     return( buffer[siguiente] );
     }

  // Método para añadir letras al buffer
  public synchronized void lanzar( char c ) {
     // Espera hasta que haya sitio para otra letra
     while( estaLlena == true )
        {
        try {
           wait(); // Se sale cuando estaLlena cambia a false
        } catch( InterruptedException e ) {
           ;
           }
        }
     // Añade una letra en el primer lugar disponible
     buffer[siguiente] = c;
     // Cambia al siguiente lugar disponible
     siguiente++;
     // Comprueba si el buffer está lleno
     if( siguiente == 6 )
        estaLlena = true;
     estaVacia = false;
     notify();
     }
  }
```

En la clase **Tuberia** se pueden observar dos características importantes: los miembros dato (buffer[]) son privados, y los métodos de acceso (*lanzar()* y *recoger()*) son sincronizados.

Aquí se observa que la variable estaVacia es un semáforo, como los de toda la vida. La naturaleza privada de los datos evita que el productor y el consumidor accedan directamente a éstos. Si se permitiese el acceso directo de ambos hilos de ejecución a los datos, se podrían producir problemas; por ejemplo, si el consumidor intenta retirar datos de un buffer vacío, obtendrá excepciones innecesarias, o se bloqueará el proceso.

Los métodos sincronizados de acceso impiden que los productores y consumidores corrompan un objeto compartido. Mientras el productor está añadiendo una letra a la tubería, el consumidor no la puede retirar y viceversa. Esta sincronización es vital para mantener la integridad de cualquier objeto compartido.

No sería lo mismo sincronizar la clase en vez de los métodos, porque esto significaría que nadie puede acceder a las variables de la clase en paralelo, mientras que al sincronizar los métodos, sí pueden acceder a todas las variables que están fuera de los métodos que pertenecen a la clase.

Se pueden sincronizar incluso variables, para realizar alguna acción determinada sobre ellas, por ejemplo:

```
sincronized( p ) {
   // aquí se colocaría el código
   // los threads que estén intentando acceder a p se pararán
   // y generarán una InterruptedException
   }
```

El método *notify()* al final de cada método de acceso avisa a cualquier proceso que esté esperando por el objeto, entonces el proceso que ha estado esperando intentará acceder de nuevo al objeto. En el método *wait()* se hace que el hilo se quede a la espera de que le llegue un *notify()*, ya sea enviado por el hilo de ejecución o por el sistema. Ahora que ya se dispone de un productor, un consumidor y un objeto compartido, se necesita una aplicación que arranque los hilos y que consiga que todos hablen con el mismo objeto que están compartiendo. Esto es lo que hace el siguiente trozo de código, del fuente java1007.java:

```
class java1007 {
   public static void main( String args[] ) {
      Tuberia t = new Tuberia();
      Productor p = new Productor( t );
      Consumidor c = new Consumidor( t );

      p.start();
      c.start();
      }
   }
```

Compilando y ejecutando esta aplicación, se podrá observar en modelo que se ha diseñado en pleno funcionamiento.