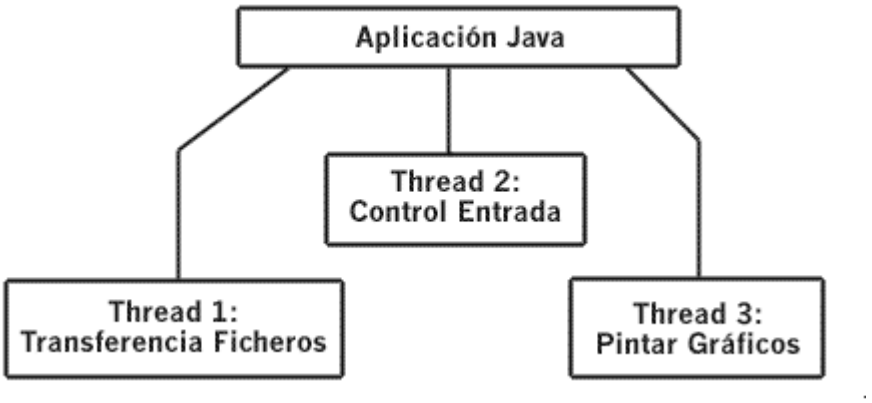
http://www.itlp.edu.mx/web/java/Tutorial%20de%20Java/Intro/tabla.html

Considerando el entorno *multithread*, 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 thread 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 threads 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 threads se parecen en su funcionamiento a lo que muestra la figura siguiente:



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 nuestra 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 thread.

Programas de flujo múltiple

En nuestra aplicación de saludo, no vemos el thread que ejecuta nuestro programa. Sin embargo, Java posibilita la creación y control de threads explícitamente. La utilización de 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 threads, 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 threads en Java. Habrá observado que dos applet 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 threads, sino que el navegador es multithreaded.

Las aplicaciones (y applets) multithreaded 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 thread para cada subtarea.

Mientras que los programas de flujo único pueden realizar su tarea ejecutando las subtareas secuencialmente, un programa multithreaded 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 nuestro programa de saludo creando tres threads individuales, que imprimen cada uno de ellos su propio mensaje de saludo, MultiHola.java:

```
// Definimos unos sencillos threads. 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();
        }
    }
```

Y ya mßs como espectáculo que otra cosa, aunque también podemos tomarlo por el lado ilustrativo, vemos a continuación la elecución del applet Figuras.java que muestra un montón de círculos, cada uno de ellos ejecutándose en un *thread* diferente y con distinta prioridad cada uno de ellos. La clase **Circulo** es la que se utiliza para lanzarla todas las veces que se quiere, de tal forma que cada uno de los círculos presentes en el applet son instancias de la misma clase **Circulo**.

Creación de un Thread

Hay dos modos de conseguir threads en Java. Una es implementando la interface **Runnable**, la otra es extender la clase **Thread**.

La implementación de la interface **Runnable** es la forma habitual de crear threads. Las 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. La interface define el trabajo y la clase, o clases, que implementan la interface realizan ese trabajo. Los diferentes grupos de clases que implementen la interface tendrán que seguir las mismas reglas de funcionamiento.

Hay una cuantas diferencias entre interface y clase. Primero, una interface 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, una interface no puede implementar cualquier método. Una clase que implemente una interface debe implementar todos los métodos definidos en esa interface. Una interface tiene la posibilidad de poder extenderse de otras interfaces y, al contrario que las clases, puede extenderse de múltiples interfaces. Además, una interface no puede ser instanciada 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 thread 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 sobrecarga 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 thread. Además, el método abstracto *run()* está definido en la interface **Runnable** tiene que ser implementado. La única diferencia entre los dos métodos es que este último es mucho más flexible. En el ejemplo anterior, todavía tenemos oportunidad de extender la clase **MiThread**, si fuese necesario. La mayoría de las clases creadas que necesiten ejecutarse como un thread , implementarán la interface **Runnable**, ya que probablemente extenderán alguna de su funcionalidad a otras clases.

No pensar que la interface **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 vemos los

fuentes de Java, podremos 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 la interface **Runnable**. Como se ve, una interface 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** nos 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 la interface **Runnable**. *tarea.run()* se asegura de que la clase con que trabaja (la clase que va a ejecutarse como un thread) 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 thread.

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 threads. La línea de código:

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

crea un nuevo thread. Los dos argumentos pasados representan el nombre del thread y el tiempo que queremos que espere antes de imprimir el mensaje.

Al tener control directo sobre los threads, tenemos que arrancarlos explícitamente. En nuestro ejemplo con:

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

start(), en realidad es un método oculto en el thread que llama al método *run()*.

Manipulación de un Thread

Si todo fue bien en la creación del thread, `t1` debería contener un thread válido, que controlaremos en el método *run()*.

Una vez dentro de *run()*, podemos comenzar las sentencias de ejecución como en otros programas. *run()* sirve como rutina *main()* para los threads; cuando *run()* termina, también lo hace el thread. Todo lo que queramos que haga el thread ha de estar dentro de *run()*, por eso cuando decimos que un método es Runnable, nos obliga a escribir un método *run()*.

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

```
sleep( retardo );
```

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

Suspensión de un Thread

Puede resultar útil suspender la ejecución de un thread sin marcar un límite de tiempo. Si, por ejemplo, está construyendo un applet con un thread de animación, 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 thread se puede utilizar el método *suspend()*.

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

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

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

Parada de un Thread

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

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

Esta llamada no destruye el thread, 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 thread, el objeto thread (creado con *new*) quedará marcado para eliminarlo y el *garbage collector* se encargará de liberar la memoria que utilizaba.

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

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

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

Este método devolverá *true* en caso de que el thread `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.

Ahora que ya hemos visto por encima como se arrancan, paran y manipulan threads, vamos a mostrar un ejemplo un poco más gráfico, se trata de un contador, cuyo código (App1Thread.java) es el siguiente:

```
import java.awt.*;
import java.applet.Applet;

public class App1Thread extends Applet implements Runnable {
    Thread t;
    int contador;

    public void init() {
        contador = 0;
        t = new Thread( this );
        t.start();
        }

    public void run() {
        while( true )
            {
            contador++;
            repaint();
            try {
                t.sleep( 10 );
            } catch( InterruptedException e ) {
                ;
                };
```

```
            }
        }

    public boolean mouseDown( Event evt,int x,int y ) {
        t.stop();
        return( true );
        }

    public void paint( Graphics g ) {
        g.drawString( Integer.toString( contador ),10,10 );
        System.out.println( "Contador = "+contador );
        }

    public void stop() {
        t.stop();
        }
    }
```

Este applet 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, nos revelará su verdadera identidad.

En este caso, la clase **App1Thread** 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 thread ya conocerá al objeto que va a correr. En este caso `this` es una referencia a *App1Thread*. Después de que hayamos creado el thread, necesitamos arrancarlo. La llamada a *start()*, llamará a su vez al método *run()* de nuestra clase, es decir, a *App1Thread.run()*. La llamada a *start()* retornará con éxito y el thread 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 thread 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 thread, porque sino, el thread consumirá todo el tiempo de la CPU para su proceso y no permitirá que entren otros métodos de otros threads a ejecutarse. Otra forma de detener la ejecución del thread es hacer una llamada al método *stop()*. En el contador, el thread 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,

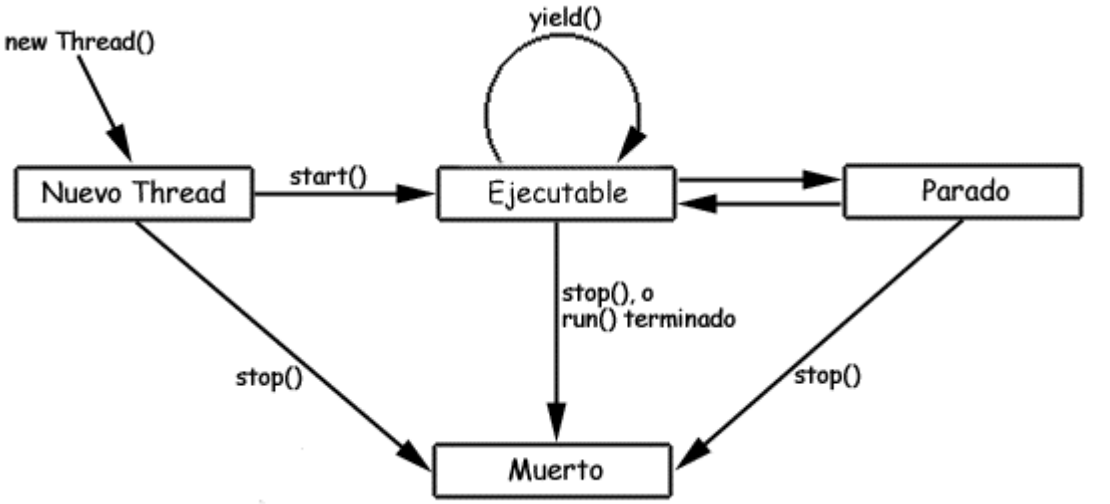
sino que el sistema operativo encolará las peticiones y las que sean sucesivas las convertirán en un único refresco. Así, mientras los refescos se van encolando, la variable `contador` se estará todavía incrementando, pero no se visualiza en pantalla.

Una vez que se para un thread, ya no se puede rearrancar con el comando *start()*, debido a que *stop()* concluirá la ejecución del thread. Por ello, en ver de parar el thread, lo que podemos hacer es dormirlo, llamando al método *sleep()*. El thread 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 thread reanude su ejecución ante la presencia de ciertos eventos. En estos casos, el método *suspend()* permite que cese la ejecución del thread y el método *resume()* permite que un método suspendido reanude su ejecución. En la siguiente versión de nuestra clase **contador**, App2Thread.java, modificamos el applet para que utilice los métodos *suspend()* y *resume()*:

```
public class App2Thread extends Applet implements Runnable {
    Thread t;
    int contador;
    boolean suspendido;

    ...

    public boolean mouseDown( Event evt,int x,int y ) {
        if( suspendido )
            t.resume();
        else
            t.suspend();
        suspendido = !suspendido;

        return( true );
        }
    ...
```

Para controlar el estado del applet, hemos 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*.

Durante el ciclo de vida de un thread, 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 thread .



Nuevo Thread

La siguiente sentencia crea un nuevo thread pero no lo arranca, lo deja en el estado de "Nuevo Thread":

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

Cuando un thread 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 veamos las dos línea 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 thread puede ejecutarse, lo incorpora a la lista de procesos disponibles para ejecución del sistema y llama al método *run()* del thread. En este momento nos encontramos en el estado "Ejecutable" del diagrama. Y este estado es Ejecutable y no En Ejecución, porque cuando el thread está aquí no esta corriendo. Muchos ordenadores tienen solamente un procesador lo que hace imposible que todos los threads 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 threads que se encuentran en la lista. Sin embargo, para nuestros propósitos, 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 nosotros es que todos los procesos se ejecutan al mismo tiempo.

Cuando el thread 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 thread entra en estado "Parado" cuando alguien llama al método *suspend()*, cuando se llama al método *sleep()*, cuando el thread está bloqueado en un proceso de entrada/salida o cuando el thread utiliza su método *wait()* para esperar a que se cumpla una determinada condición. Cuando ocurra cualquiera de las cuatro cosas anteriores, el thread 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 thread 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 thread 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 thread durmiendo no serviría para nada.

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

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

Muerto

Un thread se puede morir de dos formas: por causas naturales o porque lo maten (con *stop()*). Un thread 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 thread 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 thread, 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 thread `MiThread`, lo dormimos 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 thread que quiere detener. Así, cuando un thread es parado de este modo, muere asíncronamente. El thread morirá en el momento en que reciba la excepción ThreadDeath.

Los applets utilizarán el método *stop()* para matar a todos sus threads 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()

La interface de programación de la clase **Thread** incluye el método *isAlive()*, que devuelve *true* si el thread 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 nos devuelve *true*, sabemos que el thread se encuentra en estado "Ejecutable" o "Parado". No se puede diferenciar entre "Nuevo Thread" y "Muerto", ni entre un thread "Ejecutable" o "Parado".

Java tiene un **Scheduler**, una lista de procesos, que monitoriza todos los threads 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 threads que el scheduler identifica en este proceso de decisión. Una, la más importante, es la prioridad del thread; la otra, es el indicador de demonio. La regla básica del scheduler es que si solamente hay threads demonio ejecutándose, la *Máquina Virtual Java* (JVM) concluirá. Los nuevos threads heredan la prioridad y el indicador de demonio de los threads que los han creado. El scheduler determina qué threads deberán ejecutarse comprobando la prioridad de todos los threads, 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 threads que están corriendo en el sistema. El scheduler decide cual será el siguiente thread a ejecutarse y llama a *resume()* para darle vida durante un período fijo de tiempo. Cuando el thread ha estado en ejecución ese período de tiempo, se llama a *suspend()* y el siguiente thread en la lista de procesos será relanzado (*resume()*). Los schedulers no-preemtivos deciden que thread debe correr y lo ejecutan hasta que concluye. El thread tiene control total sobre el sistema mientras esté en ejecución. El método *yield()* es la forma en que un thread fuerza al scheduler a comenzar la ejecución de otro thread que esté esperando. Dependiendo del sistema en que esté corriendo Java, el scheduler será preemtivo o no-preemptivo.

En el siguiente ejemplo, SchThread.java, mostramos la ejecución de dos threads con diferentes prioridades. Un thread se ejecuta a prioridad más baja que el otro.

Los threads incrementarán sus contadores hasta que el thread que tiene prioridad más alta alcance al contador que corresponde a la tarea con ejecución más lenta.

Prioridades

El scheduler determina el thread 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 thread es *Thread.NORM_PRIORITY*, que tiene asignado un valor de 5. Hay otras dos variables estáticas disponibles, que son *Thread.MIN_PRORITY*, fijada a 1, y *Thread.MAX_PRIORITY*, aque tiene un valor de 10. El método *getPriority()* puede utilizarse para conocer el valor actual de la prioridad de un thread.

Threads Demonio

Los threads *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. Un ejemplo de thread demonio que está ejecutándose continuamente es el recolector de basura (*garbage collector*). Este thread, 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 thread puede fijar su indicador de demonio pasando un valor *true* al método *setDaemon()*. Si se pasa *false* a este método, el thread será devuelto por el sistema como un thread de usuario. No obstante, esto último debe realizarse antes de que se arranque el thread (*start()*).

Diferencia de threads con fork()

fork() en Unix crea un proceso hijo que tiene su propia copia de datos y código del padre. Esto funciona correctamente si estamos sobrados de memoria y disponemos de una CPU poderosa, y siempre que mantengamos 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 threads.

La multi-tarea pre-emptiva tiene sus problemas. Un thread puede interrumpir a otro en cualquier momento, de ahí lo de *pre-emptive*. Imaginarse lo que pasaría si un thread está escribiendo en un array, mientras otro thread 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 threads 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.