```
public void update (Graphics.g) {
   // se comprueba si existe el objeto invisible y si sus dimensiones son correctas
   if ((graphInv==null) || (d.width!=dimInv.width) || (d.height!=dimInv.height)) {
       dimInv = d;
       // se llama al método createImage de la clase Component
       imgInv = createImage(d.width, d.height);
       // se llama al método getGraphics de la clase Image
       graphInv = imgInv.getGraphics();
   }

   // se establecen las propiedades del contexto gráfico invisible,
   //    y se dibuja sobre él
   graphInv.setColor(getBackground());
   ...
   // finalmente se hace visible la imagen invisible a partir del punto (0, 0)
   //     utilizando el propio panel como ImageObserver
   g.drawImage(imgInv, 0, 0, this);
} // fin del método update()
```

Los gráficos y las animaciones son particularmente útiles en las applets. El ***Tutorial*** de ***Sun*** tiene un ejemplo (un ***applet***) completamente explicado y desarrollado sobre las animaciones y los distintos métodos de eliminar el flicker o parpadeo.

6. THREADS: PROGRAMAS MULTITAREA

Los procesadores y los Sistemas Operativos modernos permiten la ***multitarea***, es decir, la realización simultánea de dos o más actividades (al menos aparentemente). En la realidad, un ordenador con una sola CPU no puede realizar dos actividades a la vez. Sin embargo los Sistemas Operativos actuales son capaces de ejecutar varios programas "simultáneamente" aunque sólo se disponga de una CPU: reparten el tiempo entre dos (o más) actividades, o bien utilizan los tiempos muertos de una actividad (por ejemplo, operaciones de lectura de datos desde el teclado) para trabajar en la otra. En ordenadores con dos o más procesadores la multitarea es real, ya que cada procesador puede ejecutar un ***hilo*** o ***thread*** diferente. La Figura 6.1, tomada del ***Tutorial*** de ***Sun***, muestra los esquemas correspondientes a un programa con una o dos ***threads***.

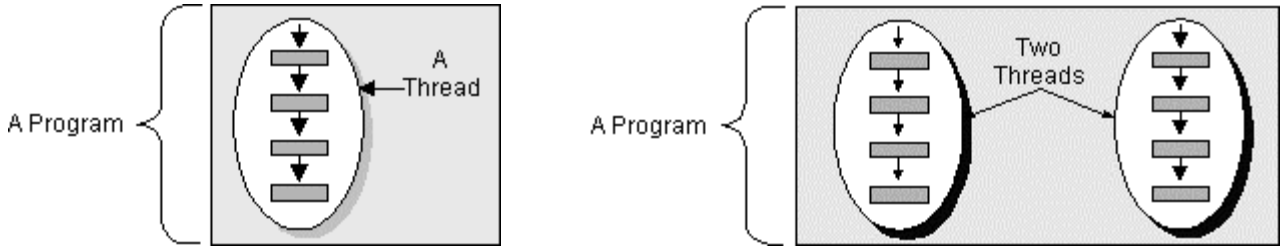

Figura 6.1. Programa con 1 y con 2 threads o hilos.

Un ***proceso*** es un programa ejecutándose de forma independiente y con un espacio propio de memoria. Un Sistema Operativo multitarea es capaz de ejecutar más de un ***proceso*** simultáneamente. Un ***thread*** o ***hilo*** es un ***flujo secuencial simple*** dentro de un ***proceso***. Un único ***proceso*** puede tener varios ***hilos*** ejecutándose. Por ejemplo el programa ***Netscape*** sería un proceso, mientras que cada una de las ventanas que se pueden tener abiertas simultáneamente trayendo páginas HTML estaría formada por al menos un ***hilo***.

Un sistema multitarea da realmente la impresión de estar haciendo varias cosas a la vez y eso es una gran ventaja para el usuario. Sin el uso de ***threads*** hay tareas que son prácticamente imposibles de ejecutar, particularmente las que tienen tiempos de espera importantes entre etapas.

Los ***threads*** o ***hilos*** de ejecución permiten organizar los recursos del ordenador de forma que pueda haber varios programas actuando en paralelo. Un ***hilo*** de ejecución puede realizar cualquier tarea que pueda realizar un programa normal y corriente. Bastará con indicar lo que tiene que hacer en el método ***run()***, que es el que define la actividad principal de las ***threads***.

Los ***threads*** pueden ser ***daemon*** o ***no daemon***. Son ***daemon*** aquellos hilos que realizan en ***background*** (en un segundo plano) servicios generales, esto es, tareas que no forman parte de la esencia del programa y que se están ejecutando mientras no finalice la aplicación. Un ***thread daemon*** podría ser por ejemplo aquél que está comprobando permanentemente si el usuario pulsa un botón. Un programa de ***Java*** finaliza cuando sólo quedan corriendo ***threads*** de tipo ***daemon***. Por defecto, y si no se indica lo contrario, los ***threads*** son del tipo ***no daemon***.

6.1 CREACIÓN DE THREADS

En ***Java*** hay dos formas de crear nuevos ***threads***. La primera de ellas consiste en crear una nueva clase que herede de la clase ***java.lang.Thread*** y sobrecargar el método ***run()*** de dicha clase. El segundo método consiste en declarar una clase que implemente la interface ***java.lang.Runnable***, la cual declarará el método ***run()***; posteriormente se crea un objeto de tipo ***Thread*** pasándole como

argumento al constructor el objeto creado de la nueva clase (la que implementa la interface ***Runnable***). Como ya se ha apuntado, tanto la clase ***Thread*** como la interface ***Runnable*** pertenecen al package ***java.lang***, por lo que no es necesario importarlas.

A continuación se presentan dos ejemplos de creación de ***threads*** con cada uno de los dos métodos citados.

6.1.1 Creación de threads derivando de la clase Thread

Considérese el siguiente ejemplo de declaración de una nueva clase:

```
public class SimpleThread extends Thread {
    // constructor
    public SimpleThread (String str) {
        super(str);
    }

    // redefinición del método run()
    public void run() {
        for(int i=0;i<10;i++)
           System.out.println("Este es el thread : " + getName());
    }
}
```

En este caso, se ha creado la clase ***SimpleThread***, que hereda de ***Thread***. En su constructor se utiliza un ***String*** (opcional) para poner nombre al nuevo ***thread*** creado, y mediante ***super()*** se llama al constructor de la super-clase ***Thread***. Asimismo, se redefine el método ***run()***, que define la principal actividad del ***thread***, para que escriba 10 veces el nombre del ***thread*** creado.

Para poner en marcha este nuevo ***thread*** se debe crear un objeto de la clase ***SimpleThread***, y llamar al método ***start()***,heredado de la super-clase ***Thread***, que se encarga de llamar a ***run()***. Por ejemplo:

```
SimpleThread miThread = new SimpleThread(“Hilo de prueba”);
miThread.start();
```

6.1.2 Creación de threads implementando la interface Runnable

Esta segunda forma también requiere que se defina el método ***run()***, pero además es necesario crear un objeto de la clase ***Thread*** para lanzar la ejecución del nuevo hilo. Al constructor de la clase ***Thread*** hay que pasarle una referencia del objeto de la clase que implementa la interface ***Runnable***. Posteriormente, cuando se ejecute el método ***start()*** del ***thread***, éste llamará al método ***run()*** definido en la nueva clase. A continuación se muestra el mismo estilo de clase que en el ejemplo anterior implementada mediante la interface ***Runnable***:

```
public class SimpleRunnable implements Runnable {
    // se crea un nombre
    String nameThread;
    // constructor
    public SimpleRunnable (String str) {
        nameThread = str;
    }
    // definición del método run()
    public void run() {
        for(int i=0;i<10;i++)
           System.out.println("Este es el thread: " + nameThread);
    }
}
```

El siguiente código crea un nuevo ***thread*** y lo ejecuta por este segundo procedimiento:

```
SimpleRunnable p = new SimpleRunnable("Hilo de prueba");
// se crea un objeto de la clase Thread pasándolo el objeto Runnable como argumento
Thread miThread = new Thread(p);
// se arranca el objeto de la clase Thread
miThread.start();
```

Este segundo método cobra especial interés con las ***applets***, ya que cualquier ***applet*** debe heredar de la clase ***java.applet.Applet***, y por lo tanto ya no puede heredar de ***Thread***. Véase el siguiente ejemplo:

```
class ThreadRunnable extends Applet implements Runnable {
    private Thread runner=null;
    // se redefine el método start() de Applet
    public void start() {
        if (runner == null) {
            runner = new Thread(this);
            runner.start(); // se llama al método start() de Thread
        }
    }
    // se redefine el método stop() de Applet
    public void stop(){
        runner = null; // se libera el objeto runner
    }
}
```

En este ejemplo, el argumento ***this*** del constructor de ***Thread*** hace referencia al objeto ***Runnable*** cuyo método ***run()*** debería ser llamado cuando el hilo ejecutado es un objeto de ***ThreadRunnable***.

La elección de una u otra forma -derivar de ***Thread*** o implementar ***Runnable***- depende del tipo de clase que se vaya a crear. Así, si la clase a utilizar ya hereda de otra clase (por ejemplo un ***applet***, que siempre hereda de ***Applet***), no quedará más remedio que implementar ***Runnable***, aunque normalmente es más sencillo heredar de ***Thread***.

6.2 CICLO DE VIDA DE UN THREAD

En el apartado anterior se ha visto cómo crear nuevos objetos que permiten incorporar en un programa la posibilidad de realizar varias tareas simultáneamente. En la Figura 6.2 (tomada del ***Tutorial*** de **Sun**) se muestran los distintos estados por los que puede pasar un ***thread*** a lo largo de su vida. Un ***thread*** puede presentar cuatro estados distintos:

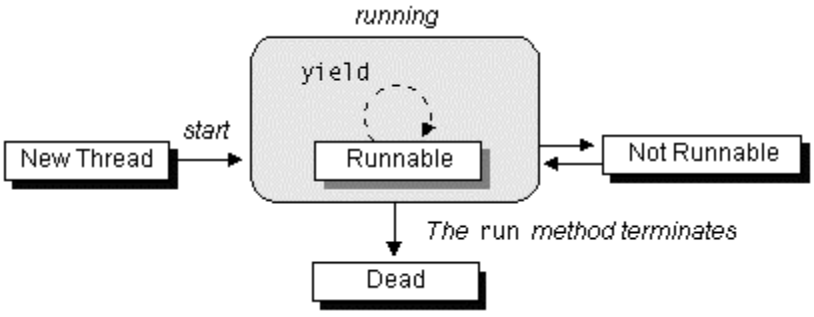

Figura 6.2. Ciclo de vida de un Thread.

1. *Nuevo* (***New***): El ***thread*** ha sido creado pero no inicializado, es decir, no se ha ejecutado todavía el método ***start()***. Se producirá un mensaje de error (***IllegalThreadStateException***) si se intenta ejecutar cualquier método de la clase ***Thread*** distinto de ***start()***.
2. *Ejecutable* (***Runnable***): El ***thread*** puede estar ejecutándose, siempre y cuando se le haya asignado un determinado tiempo de CPU. En la práctica puede no estar siendo ejecutado en un instante determinado en beneficio de otro ***thread***.
3. *Bloqueado* (***Blocked*** o ***Not Runnable***): El ***thread*** podría estar ejecutándose, pero hay alguna actividad interna suya que lo impide, como por ejemplo una espera producida por

una operación de escritura o lectura de datos por teclado (E/S). Si un ***thread*** está en este estado, no se le asigna tiempo de CPU.

4. *Muerto* (***Dead***): La forma habitual de que un ***thread*** muera es finalizando el método ***run()***. También puede llamarse al método ***stop()*** de la clase ***Thread***, aunque dicho método es considerado "peligroso" y no se debe utilizar.

A continuación se explicarán con mayor detenimiento los puntos anteriores.

6.2.1 Ejecución de un nuevo thread

La creación de un nuevo ***thread*** no implica necesariamente que se empiece a ejecutar algo. Hace falta iniciarlo con el método ***start()***, ya que de otro modo, cuando se intenta ejecutar cualquier método del ***thread*** -distinto del método ***start()***- se obtiene en tiempo de ejecución el error ***IllegalThreadStateException***.

El método ***start()*** se encarga de llamar al método ***run()*** de la clase ***Thread***. Si el nuevo ***thread*** se ha creado heredando de la clase ***Thread*** la nueva clase deberá redefinirir el método ***run()*** heredado. En el caso de utilizar una clase que implemente la interface ***Runnable***, el método ***run()*** de la clase ***Thread*** se ocupa de llamar al método ***run()*** de la nueva clase (véase el Apartado 6.1.2, en la página 117).

Una vez que el método ***start()*** ha sido llamado, se puede decir ya que el ***thread*** está "corriendo" (***running***), lo cual no quiere decir que se esté ejecutando en todo momento, pues ese ***thread*** tiene que compartir el tiempo de la CPU con los demás ***threads*** que también estén ***running***. Por eso más bien se dice que dicha ***thread*** es ***runnable***.

6.2.2 Detener un Thread temporalmente: Runnable - Not Runnable

El sistema operativo se ocupa de asignar tiempos de CPU a los distintos ***threads*** que se estén ejecutando simultáneamente. Aun en el caso de disponer de un ordenador con más de un procesador (2 ó más CPUs), el número de ***threads*** simultáneos suele siempre superar el número de CPUs, por lo que se debe repartir el tiempo de forma que parezca que todos los procesos corren a la vez (quizás más lentamente), aun cuando sólo unos pocos pueden estar ejecutándose en un instante de tiempo.

Los tiempos de CPU que el sistema continuamente asigna a los distintos ***threads*** en estado ***runnable*** se utilizan en ejecutar el método ***run()*** de cada ***thread***. Por diversos motivos, un ***thread*** puede en un determinado momento renunciar "voluntariamente" a su tiempo de CPU y otorgárselo al sistema para que se lo asigne a otro ***thread***. Esta "renuncia" se realiza mediante el método ***yield()***. Es importante que este método sea utilizado por las actividades que tienden a "monopolizar" la CPU. El método ***yield()*** viene a indicar que en ese momento no es muy importante para ese ***thread*** el ejecutarse continuamente y por lo tanto tener ocupada la CPU. En caso de que ningún ***thread*** esté requiriendo la CPU para una actividad muy intensiva, el sistema volverá casi de inmediato a asignar nuevo tiempo al ***thread*** que fue "generoso" con los demás. Por ejemplo, en un Pentium II 400 Mhz es posible llegar a más de medio millón de llamadas por segundo al método ***yield()***, dentro del método ***run()***, lo que significa que llamar al método ***yield()*** apenas detiene al ***thread***, sino que sólo ofrece el control de la CPU para que el sistema decida si hay alguna otra tarea que tenga mayor prioridad.

Si lo que se desea es parar o bloquear temporalmente un ***thread*** (pasar al estado ***Not Runnable***), existen varias formas de hacerlo:

1. Ejecutando el método ***sleep()*** de la clase ***Thread***. Esto detiene el ***thread*** un tiempo pre-establecido. De ordinario el método ***sleep()*** se llama desde el método ***run()***.
2. Ejecutando el método ***wait()*** heredado de la clase ***Object***, a la espera de que suceda algo que es necesario para poder continuar. El ***thread*** volverá nuevamente a la situación de ***runnable*** mediante los métodos ***notify()*** o ***notifyAll()***, que se deberán ejecutar cuando cesa la condición que tiene detenido al thread (ver Apartado 6.3, en la página 121).
3. Cuando el ***thread*** está esperando para realizar operaciones de Entrada/Salida o Input/Output (E/S ó I/O).
4. Cuando el ***thread*** está tratando de llamar a un método ***synchronized*** de un objeto, y dicho objeto está bloqueado por otro ***thread*** (véase el Apartado 6.3)

Un ***thread*** pasa automáticamente del estado ***Not Runnable*** a ***Runnable*** cuando cesa alguna de las condiciones anteriores o cuando se llama a ***notify()*** o ***notifyAll()***.

La clase ***Thread*** dispone también de un método ***stop()***, pero ***no se debe utilizar*** ya que puede provocar bloqueos del programa (***deadlock***). Hay una última posibilidad para detener un ***thread***, que consiste en ejecutar el método ***suspend()***. El ***thread*** volverá a ser ejecutable de nuevo ejecutando el método ***resume()***. Esta última forma también se desaconseja, por razones similares a la utilización del método ***stop()***.

El método ***sleep()*** de la clase ***Thread*** recibe como argumento el tiempo en *milisegundos* que ha de permanecer detenido. Adicionalmente, se puede incluir un número entero con un tiempo adicional en *nanosegundos*. Las declaraciones de estos métodos son las siguientes:

```
public static void sleep(long millis) throws InterruptedException
public static void sleep(long millis, int nanosecons) throws InterruptedException
```

Considérese el siguiente ejemplo:

```
System.out.println ("Contador de segundos");
int count=0;
public void run () {
    try {
        sleep(1000);
        System.out.println(count++);
    } catch (InterruptedException e){}
}
```

Se observa que el método ***sleep()*** puede lanzar una ***InterruptedException*** que ha de ser capturada. Así se ha hecho en este ejemplo, aunque luego no se gestiona esa excepción.

La forma preferible de detener temporalmente un ***thread*** es la utilización conjunta de los métodos ***wait()*** y ***notifyAll()***. La principal ventaja del método ***wait()*** frente a los métodos anteriormente descritos es que libera el bloqueo del objeto. por lo que el resto de threads que se encuentran esperando para actuar sobre dicho objeto pueden llamar a sus métodos. Hay dos formas de llamar a ***wait()***:

1. Indicando el tiempo máximo que debe estar parado (en *milisegundos* y con la opción de indicar también *nanosegundos*), de forma análoga a ***sleep()***. A diferencia del método ***sleep()***, que simplemente detiene el ***thread*** el tiempo indicado, el método ***wait()*** establece el tiempo máximo que debe estar parado. Si en ese plazo se ejecutan los métodos ***notify()*** o ***notifyAll()*** que indican la liberación de los objetos bloqueados, el ***thread*** continuará sin esperar a concluir el tiempo indicado. Las dos declaraciones del método ***wait()*** son como siguen:

```
public final void wait(long timeout) throws InterruptedException
public final void wait(long timeout, int nanos) throws InterruptedException
```

2. Sin argumentos, en cuyo caso el ***thread*** permanece parado hasta que sea reinicializado explícitamente mediante los métodos ***notify()*** o ***notifyAll()***.

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

Los métodos ***wait()*** y ***notify()*** han de estar incluidas en un método ***synchronized,*** ya que de otra forma se obtendrá una excepción del tipo ***IllegalMonitorStateException*** en tiempo de ejecución. El uso típico de ***wait()*** es el de esperar a que se cumpla alguna determinada condición, ajena al propio ***thread***. Cuando ésta se cumpla, se utilizará el método ***notifyAll()*** para avisar a los distintos ***threads*** que pueden utilizar el objeto. Estos nuevos conceptos se explican con más profundidad en el Apartado 6.3.

6.2.3 Finalizar un Thread

Un ***thread*** finaliza cuando el método ***run()*** devuelve el control, por haber terminado lo que tenía que hacer (por ejemplo, un bucle ***for*** que se ejecuta un número determinado de veces) o por haberse dejado de cumplir una condición (por ejemplo, por un bucle ***while*** en el método ***run()***). Es habitual poner las siguientes sentencias en el caso de ***Applets Runnables***:

```
public class MyApplet extends Applet implements Runnable {
    // se crea una referencia tipo Thread
    private Thread AppletThread;
    ...
    // método start() del Applet
    public void start() {
        if(AppletThread == null){          // si no tiene un objeto Thread asociado
           AppletThread = new Thread(this, "El propio Applet");
           AppletThread.start();           // se arranca el thread y llama a run()
        }
    }
    // método stop() del Applet
    public void stop() {
        AppletThread = null;               // iguala la referencia a null
    }
    // método run() por implementar Runnable
    public void run() {
        Thread myThread = Thread.currentThread();
        while (myThread == AppletThread) { // hasta que se ejecute stop() de Thread
           ...                             // código a ejecutar
        }
    }
} // fin de la clase MyApplet
```

donde ***AppletThread*** es el ***thread*** que ejecuta el método ***run()*** MyApplet. Para finalizar el thread basta poner la referencia ***AppletThread*** a ***null***. Esto se consigue en el ejemplo con el método ***stop()*** del ***applet*** (distinto del método ***stop()*** de la clase ***Thread***, que no conviene utilizar).

Para saber si un ***thread*** está “vivo” o no, es útil el método ***isAlive()*** de la clase ***Thread***, que devuelve ***true*** si el ***thread*** ha sido inicializado y no parado, y ***false*** si el ***thread*** es todavía nuevo (no ha sido inicializado) o ha finalizado.

6.3 SINCRONIZACIÓN

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

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

```
public synchronized void metodoSincronizado() {
   ...// accediendo por ejemplo a las variables de un objeto
   ...
}
```

La ***sincronización*** previene las interferencias solamente sobre un tipo de recurso: la memoria reservada para un objeto. Cuando se prevea que unas determinadas variables de una clase pueden tener problemas de sincronización, se deberán declarar como ***private*** (o ***protected***). De esta forma sólo estarán accesibles a través de métodos de la clase, que deberán estar ***sincronizados***.

Es muy importante tener en cuenta que si se sincronizan algunos métodos de un objeto pero otros no, el programa puede no funcionar correctamente. La razón es que los métodos no sincronizados pueden acceder libremente a las variables miembro, ignorando el bloqueo del objeto. Sólo los métodos sincronizados comprueban si un objeto está bloqueado. Por lo tanto, todos los métodos que accedan a un recurso compartido deben ser declarados ***synchronized***. De esta forma, si algún método accede a un determinado recurso, ***Java*** bloquea dicho recurso, de forma que el resto de ***threads*** no puedan acceder al mismo hasta que el primero en acceder termine de realizar su tarea. ***Bloquear un recurso u objeto*** significa que sobre ese objeto no pueden actuar simultáneamente dos ***métodos sincronizados***.

Existen dos niveles de bloqueo de un recurso. El primero es ***a nivel de objetos***, mientras que el segundo es ***a nivel de clases***. El primero se consigue declarando todos los métodos de una clase como ***synchronized***. Cuando se ejecuta un método ***synchronized*** sobre un objeto concreto, el sistema bloquea dicho objeto, de forma que si otro ***thread*** intenta ejecutar algún método sincronizado de ese objeto, este segundo método se mantendrá a la espera hasta que finalice el anterior (y desbloquee por lo tanto el objeto). Si existen varios objetos de una misma clase, como los bloqueos se producen a nivel de objeto, es posible tener distintos ***threads*** ejecutando métodos sobre diversos objetos de una misma clase.

El bloqueo de recursos ***a nivel de clases*** se corresponde con los ***métodos de clase*** o ***static***, y por lo tanto con las ***variables de clase*** o ***static***. Si lo que se desea es conseguir que un método bloquee simultáneamente una clase entera, es decir todos los objetos creados de una clase, es necesario declarar este método como ***synchronized static***. Durante la ejecución de un método declarado de esta segunda forma ningún método sincronizado tendrá acceso a ningún objeto de la clase bloqueada.

La sincronización puede ser problemática y generar errores. Un ***thread*** podría bloquear un determinado recurso de forma indefinida, impidiendo que el resto de ***threads*** accedieran al mismo. Para evitar esto último, habrá que utilizar la sincronización sólo donde sea estrictamente necesario.

Es necesario tener presente que si dentro un método sincronizado se utiliza el método ***sleep()*** de la clase ***Thread***, el objeto bloqueado permanecerá en ese estado durante el tiempo indicado en el argumento de dicho método. Esto implica que otros ***threads*** no podrán acceder a ese objeto durante ese tiempo, aunque en realidad no exista peligro de simultaneidad ya que durante ese tiempo el thread que mantiene bloqueado el objeto no realizará cambios. Para evitarlo es conveniente sustituir ***sleep()*** por el método ***wait()*** de la clase ***java.lang.Object*** heredado automáticamente por todas las clases. Cuando se llama al método ***wait()*** (siempre debe hacerse desde un método o bloque

synchronized) se libera el bloqueo del objeto y por lo tanto es posible continuar utilizando ese objeto a través de métodos sincronizados. El método ***wait()*** detiene el ***thread*** hasta que se llame al método ***notify()*** o ***notifyAll()*** del objeto, o finalice el tiempo indicado como argumento del método ***wait()***. El método ***unObjeto.notify()*** lanza una señal indicando al sistema que puede activar uno de los ***threads*** que se encuentren bloqueados esperando para acceder al objeto ***unObjeto***. El método ***notifyAll()*** lanza una señal a todos los ***threads*** que están esperando la liberación del objeto.

Los métodos ***notify()*** y ***notifyAll()*** deben ser llamados desde el ***thread*** que tiene bloqueado el objeto para activar el resto de threads que están esperando la liberación de un objeto. Un ***thread*** se convierte en propietario del bloqueo de un objeto ejecutando un método sincronizado del objeto. Los bloqueos de tipo clase, se consiguen ejecutando un ***método de clase sincronizado*** (***synchronized static***). Véanse las dos funciones siguientes, de las que ***put()*** inserta un dato y ***get()*** lo recoge:

```
public synchronized int get() {
    while (available == false) {
        try {
            // Espera a que put() asigne el valor y lo comunique con notify()
            wait();
        } catch (InterruptedException e) { }
    }
    available = false;
    // notifica que el valor ha sido leído
    notifyAll();
    // devuelve el valor
    return contents;
}
public synchronized void put(int value) {
    while (available == true) {
        try {
            // Espera a que get() lea el valor disponible antes de darle otro
            wait();
        } catch (InterruptedException e) { }
    }
    // ofrece un nuevo valor y lo declara disponible
    contents = value;
    available = true;
    // notifica que el valor ha sido cambiado
    notifyAll();
}
```

El bucle ***while*** de la función ***get()*** continúa ejecutándose (`avalaible == false`) hasta que el método ***put()*** haya suministrado un nuevo valor y lo indique con `avalaible = true`. En cada iteración del ***while*** la función ***wait()*** hace que el hilo que ejecuta el método ***get()*** se detenga hasta que se produzca un mensaje de que algo ha sido cambiado (en este caso con el método ***notifAll()*** ejecutado por ***put()***). El método ***put()*** funciona de forma similar.

Existe también la posibilidad de sincronizar una parte del código de un método sin necesidad de mantener bloqueado el objeto desde el comienzo hasta el final del método. Para ello se utiliza la palabra clave ***syncronized*** indicando entre paréntesis el objeto que se desea sincronizar (`synchronized(objetoASincronizar)`). Por ejemplo si se desea sincronizar el propio ***thread*** en una parte del método ***run()***, el código podría ser:

```
public void run() {
    while(true) {
        ...
        syncronized(this) { // El objeto a sincronizar es el propio thread
            ...             // Código sincronizado
        }
        try {
            sleep(500); // Se detiene el thread durante 0.5 segundos pero el objeto
                        // es accesible por otros threads al no estar sincronizado
        } catch(InterruptedException e) {}
```

```
    }
}
```

Un ***thread*** puede llamar a un método sincronizado de un objeto para el cual ya posee el bloqueo, volviendo a adquirir el bloqueo. Por ejemplo:

```
public class VolverAAdquirir {
    public synchronized void a() {
        b();
        System.out.println("Estoy en a()");
    }
    public synchronized void b() {
        System.out.println("Estoy en b()");
    }
}
```

El anterior ejemplo obtendrá como resultado:

```
Estoy en b()
Estoy en a()
```

debido a que se ha podido acceder al objeto con el método ***b()*** al ser el ***thread*** que ejecuta el método ***a()*** "propietario" con anterioridad del bloqueo del objeto.

La sincronización es un proceso que lleva bastante tiempo a la CPU, luego se debe minimizar su uso, ya que el programa será más lento cuanta más sincronización incorpore.

6.4 PRIORIDADES

Con el fin de conseguir una correcta ejecución de un programa se establecen ***prioridades*** en los ***threads***, de forma que se produzca un reparto más eficiente de los recursos disponibles. Así, en un determinado momento, interesará que un determinado proceso acabe lo antes posible sus cálculos, de forma que habrá que otorgarle más recursos (más tiempo de CPU). Esto no significa que el resto de procesos no requieran tiempo de CPU, sino que necesitarán menos. La forma de llevar a cabo esto es gracias a las prioridades.

Cuando se crea un nuevo ***thread***, éste hereda la prioridad del ***thread*** desde el que ha sido inicializado. Las prioridades viene definidas por variables miembro de la clase ***Thread***, que toman valores enteros que oscilan entre la máxima prioridad ***MAX_PRIORITY*** (normalmente tiene el valor 10) y la mínima prioridad ***MIN_PRIORITY*** (valor 1), siendo la prioridad por defecto ***NORM_PRIORITY*** (valor 5). Para modificar la prioridad de un ***thread*** se utiliza el método ***setPriority()***. Se obtiene su valor con ***getPriority()***.

El algoritmo de distribución de recursos en ***Java*** escoge por norma general aquel ***thread*** que tiene una prioridad mayor, aunque no siempre ocurra así, para evitar que algunos procesos queden "dormidos". Cuando hay dos o más ***threads*** de la misma prioridad (y además, dicha prioridad es la más elevada), el sistema no establecerá prioridades entre los mismos, y los ejecutará alternativamente dependiendo del sistema operativo en el que esté siendo ejecutado. Si dicho SO soporta el "time-slicing" (reparto del tiempo de CPU), como por ejemplo lo hace Windows 95/98/NT, los ***threads*** serán ejecutados alternativamente.

Un ***thread*** puede en un determinado momento renunciar a su tiempo de CPU y otorgárselo a otro ***thread*** de la misma prioridad, mediante el método ***yield()***, aunque en ningún caso a un ***thread*** de prioridad inferior.

6.5 GRUPOS DE THREADS

Todo hilo de ***Java*** debe formar parte de un grupo de hilos (***ThreadGroup***). Puede pertenecer al grupo por defecto o a uno explícitamente creado por el usuario. Los grupos de ***threads*** proporcionan una forma sencilla de manejar múltiples ***threads*** como un solo objeto. Así, por ejemplo es posible parar varios ***threads*** con una sola llamada al método correspondiente. Una vez que un ***thread*** ha sido asociado a un ***threadgroup***, no puede cambiar de grupo.

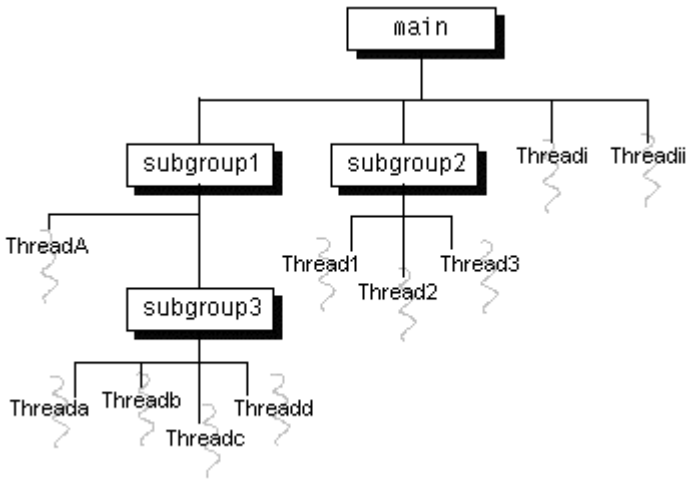

Figura 6.3. Representación de los grupos de Threads.

Cuando se arranca un programa, el sistema crea un ***ThreadGroup*** llamado ***main***. Si en la creación de un nuevo ***thread*** no se especifica a qué grupo pertenece, automáticamente pasa a pertenecer al ***threadgroup*** del ***thread*** desde el que ha sido creado (conocido como ***current thread group*** y ***current thread***, respectivamente). Si en dicho programa no se crea ningún ***ThreadGroup*** adicional, todos los ***threads*** creados pertenecerán al grupo ***main*** (en este grupo se encuentra el método ***main()***). La Figura 6.3 presenta una posible distribución de ***threads*** distribuidos en grupos de ***threads***.

Para conseguir que un **thread** pertenezca a un grupo concreto, hay que indicarlo al crear el nuevo ***thread***, según uno de los siguientes constructores:

```
public Thread (ThreadGroup grupo, Runnable destino)
public Thread (ThreadGroup grupo, String nombre)
public Thread (ThreadGroup grupo, Runnable destino, String nombre)
```

A su vez, un ***ThreadGroup*** debe pertenecer a otro ***ThreadGroup***. Como ocurría en el caso anterior, si no se especifica ninguno, el nuevo grupo pertenecerá al ***ThreadGroup*** desde el que ha sido creado (por defecto al grupo ***main***). La clase ***ThreadGroup*** tiene dos posibles constructores:

```
ThreadGroup(ThreadGroup parent, String nombre);
ThreadGroup(String name);
```

el segundo de los cuales toma como ***parent*** el ***threadgroup*** al cual pertenezca el ***thread*** desde el que se crea (***Thread.currentThread()***). Para más información acerca de estos constructores, dirigirse a la documentación del API de ***Java*** donde aparecen numerosos métodos para trabajar con ***grupos de threads*** a disposición del usuario (***getMaxPriority()***, ***setMaxPriority()***, ***getName()***, ***getParent()***, ***parentOf()***).

En la práctica los ***ThreadGroups*** no se suelen utilizar demasiado. Su uso práctico se limita a efectuar determinadas operaciones de forma más simple que de forma individual. En cualquier caso, véase el siguiente ejemplo:

```
ThreadGroup miThreadGroup = new ThreadGroup("Mi Grupo de Threads");
Thread miThread = new Thread(miThreadGroup, ”un thread para mi grupo");
```

donde se crea un grupo de ***threads*** (***miThreadGroup***) y un ***thread*** que pertenece a dicho ***grupo*** (***miThread)***.

7. APPLETS

7.1 QUÉ ES UN APPLET

Un ***applet*** es una mini-aplicación, escrita en ***Java***, que se ejecuta en un browser (***Netscape Navigator***, ***Microsoft Internet Explorer***, …) al cargar una página HTML que incluye información sobre el ***applet*** a ejecutar por medio de las ***tags*** <APPLET>... </APPLET>.

A continuación se detallan algunas características de las ***applets***:

1. Los ficheros de ***Java*** compilados (*.class) se descargan a través de la red desde un servidor de ***Web*** o servidor ***HTTP*** hasta el browser en cuya *Java Virtual Machine* se ejecutan. Pueden incluir también ficheros de imágenes y sonido.
2. Las ***applets*** no tienen ventana propia: se ejecutan en la ventana del browser (en un "***panel***").
3. Por la propia naturaleza "abierta" de Internet, las ***applets*** tienen importantes restricciones de seguridad, que se comprueban al llegar al browser: sólo pueden leer y escribir ficheros en el servidor del que han venido, sólo pueden acceder a una limitada información sobre el ordenador en el que se están ejecutando, etc. Con ciertas condiciones, las ***applets*** "de confianza" (***trusted applets***) pueden pasar por encima de estas restricciones.

Aunque su entorno de ejecución es un browser, las ***applets*** se pueden probar sin necesidad de browser con la aplicación ***appletviewer*** del JDK de ***Sun***.

7.1.1 Algunas características de las applets

Las características de las ***applets*** se pueden considerar desde el punto de vista del programador y desde el del usuario. En este manual lo más importante es el punto de vista del programador:

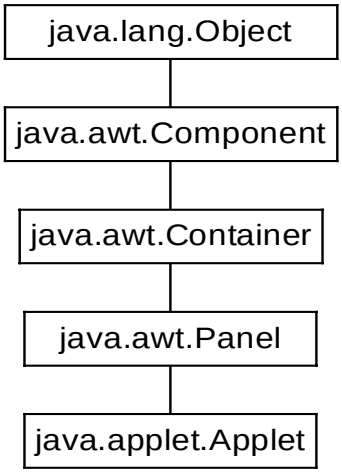

Figura 7.1. Jerarquía de clases de Applet.

- Las ***applets*** no tienen un método ***main()*** con el que comience la ejecución. El papel central de su ejecución lo asumen otros métodos que se verán posteriormente.
- Todas las ***applets*** derivan de la clase ***java.applet.Applet***. La Figura 7.1 muestra la jerarquía de clases de la que deriva la clase ***Applet***. Las ***applets*** deben redefinir ciertos métodos heredados de ***Applet*** que controlan su ejecución: ***init()***, ***start()***, ***stop()***, ***destroy()***.
- Se heredan otros muchos métodos de las super-clases de ***Applet*** que tienen que ver con la generación de interfaces gráficas de usuario (AWT). Así, los métodos gráficos se heredan de ***Component***, mientras que la capacidad de añadir componentes de interface de usuario se hereda de ***Container*** y de ***Panel***.
- Las ***applets*** también suelen redefinir ciertos métodos gráficos: los más importantes son ***paint()*** y ***update()***, heredados de ***Component*** y de ***Container***; y ***repaint()*** heredado de ***Component***.
- Las ***applets*** disponen de métodos relacionados con la obtención de información, como por ejemplo: ***getAppletInfo()***, ***getAppletContext()***, ***getParameterInfo()***, ***getParameter()***, ***getCodeBase()***, ***getDocumentBase()***, e ***isActive()***.

El método ***showStatus()*** se utiliza para mostrar información en la barra de estado del browser. Existen otros métodos relacionados con imágenes y sonido: ***getImage()***, ***getAudioClip()***, ***play()***, etc.

7.1.2 Métodos que controlan la ejecución de un applet

Los métodos que se estudian en este Apartado controlan la ejecución de las ***applets***. De ordinario el programador tiene que redefinir uno o más de estos métodos, pero no tiene que preocuparse de llamarlos: el browser se encarga de hacerlo.

7.1.2.1 Método init()

Se llama automáticamente al método ***init()*** en cuanto el browser o visualizador carga el ***applet***. Este método se ocupa de todas las tareas de inicialización, realizando las funciones del ***constructor*** (al que el browser no llama).

En ***Netscape Navigator*** se puede reinicializar un ***applet*** con ***Shift+Reload***.

7.1.2.2 Método start()

El método ***start()*** se llama automáticamente en cuanto el ***applet*** se hace visible, después de haber sido inicializada. Se llama también cada vez que el ***applet*** se hace de nuevo visible después de haber estado oculta (por dejar de estar activa esa página del browser, al cambiar el tamaño de la ventana del browser, al hacer ***reload***, etc.).

Es habitual crear ***threads*** en este método para aquellas tareas que, por el tiempo que requieren, dejarían sin recursos al ***applet*** o incluso al browser. Las animaciones y ciertas tareas a través de Internet son ejemplos de este tipo de tareas.

7.1.2.3 Método stop()

El método ***stop()*** se llama de forma automática al ocultar el ***applet*** (por haber haber dejado de estar activa la página del browser, por hacer ***reload*** o ***resize***, etc.).

Con objeto de no consumir recursos inútilmente, en este método se suelen parar las ***threads*** que estén corriendo en el ***applet***, por ejemplo para mostrar animaciones.

7.1.2.4 Método destroy()

Se llama a este método cuando el ***applet*** va a ser descargada para liberar los recursos que tenga reservados (excepto la memoria). De ordinario no es necesario redefinir este método, pues el que se hereda cumple bien con esta misión.

7.1.3 Métodos para dibujar el applet

Las ***applets*** son aplicaciones gráficas que aparecen en una zona de la ventana del browser. Por ello deben redefinir los métodos gráficos ***paint()*** y ***update()***. El método ***paint()*** se declara en la forma:

```
public void paint(Graphics g)
```

El objeto gráfico ***g*** pertenece a la clase ***java.awt.Graphics***, que siempre debe ser importada por el ***applet***. Este objeto define un contexto o estado gráfico para dibujar (métodos gráficos, colores, fonts, etc.) y es creado por el browser.

Todo el trabajo gráfico del ***applet*** (dibujo de líneas, formas gráficas, texto, etc.) se debe incluir en el método ***paint()***, porque este método es llamado cuando el ***applet*** se dibuja por primera vez y también de forma automática cada vez que el ***applet*** se debe redibujar.

En general, el programador crea el método ***paint()*** pero no lo suele llamar. Para pedir explícitamente al sistema que vuelva a dibujar el ***applet*** (por ejemplo, por haber realizado algún cambio) se utiliza el método ***repaint()***, que es más fácil de usar, pues no requiere argumentos. El método ***repaint()*** se encarga de llamar a ***paint()*** a través de ***update()***.

El método ***repaint()*** llama a ***update()***, que borra todo pintando de nuevo con el color de fondo y luego llama a ***paint()***. A veces esto produce parpadeo de pantalla o ***flickering***. Existen dos formas de evitar el ***flickering***:

1. Redefinir ***update()*** de forma que no borre toda la ventana sino sólo lo necesario.
2. Redefinir ***paint()*** y ***update()*** para utilizar ***doble buffer***.

Ambas formas fueron consideradas en los Apartados 5.6.1 y 5.6.2, en la página 114.

7.2 CÓMO INCLUIR UN APPLET EN UNA PÁGINA HTML

Para llamar a un ***applet*** desde una página HTML se utiliza la tag doble <APPLET>…</APPLET>, cuya forma general es (los elementos opcionales aparecen entre corchetes[]):

```
<APPLET CODE="miApplet.class" [CODEBASE="unURL"] [NAME="unName"]
    WIDTH="wpixels" HEIGHT="hpixels"
    [ALT="TextoAlternativo"]>
    [texto alternativo para browsers que reconocen el tag <applet> pero no pueden
           ejecutar el applet]
    [<PARAM NAME="MyName1" VALUE="valueOfMyName1">]
    [<PARAM NAME="MyName2" VALUE="valueOfMyName2">]
</APPLET>
```

El atributo NAME permite dar un nombre opcional al ***applet***, con objeto de poder comunicarse con otras ***applets*** o con otros elementos que se estén ejecutando en la misma página. El atributo ARCHIVE permite indicar uno o varios ficheros Jar o Zip (separados por comas) donde se deben buscar las clases.

A continuación se señalan otros posibles atributos de <APPLET>:

- ARCHIVE="file1, file2, file3". Se utiliza para especificar ficheros JAR y ZIP.
- ALIGN, VSPACE, HSPACE. Tienen el mismo significado que el tag IMG de HTML.

7.3 PASO DE PARÁMETROS A UN APPLET

Los tags PARAM permiten pasar diversos parámetros desde el fichero HTML al programa ***Java*** del ***applet***, de una forma análoga a la que se utiliza para pasar argumentos a ***main()***.

Cada parámetro tiene un ***nombre*** y un ***valor***. Ambos se dan en forma de ***String***, aunque el valor sea numérico. El ***applet*** recupera estos parámetros y, si es necesario, convierte los ***Strings*** en valores numéricos. El valor de los parámetros se obtienen con el siguiente método de la clase ***Applet***:

```
String getParameter(String name)
```

La conversión de ***Strings*** a los tipos primitivos se puede hacer con los métodos asociados a los ***wrappers*** que ***Java*** proporciona para dichos tipo fundamentales (Integer.parseInt(String),

Double.valueOf(String), …). Estas clases de wrappers se estudiaron en el Apartado 4.3, a partir de la página 64.

En los ***nombres*** de los parámetros no se distingue entre mayúsculas y minúsculas, pero sí en los ***valores***, ya que serán interpretados por un programa ***Java***, que sí distingue.

El programador del ***applet*** debería prever siempre unos ***valores por defecto*** para los parámetros del ***applet***, para el caso de que en la página HTML que llama al ***applet*** no se definan.

El método ***getParameterInfo()*** devuelve una matriz de ***Strings*** (*String[][]*) con información sobre cada uno de los parámetros soportados por el ***applet***: *nombre*, *tipo* y *descripción*, cada uno de ellos en un ***String***. Este método debe ser redefinido por el programador del ***applet*** y utilizado por la persona que prepara la página HTML que llama al ***applet***. En muchas ocasiones serán personas distintas, y ésta es una forma de que el programador del ***applet*** dé información al usuario.

7.4 CARGA DE APPLETS

7.4.1 Localización de ficheros

Por defecto se supone que los ficheros ****.class*** del ***applet*** están en el mismo directorio que el fichero HTML. Si el ***applet*** pertenece a un ***package***, el browser utiliza el nombre del ***package*** para construir un ***path de directorio*** relativo al directorio donde está el HTML.

El atributo CODEBASE permite definir un URL para los ficheros que continen el código y demás elementos del ***applet***. Si el directorio definido por el URL de CODEBASE es *relativo*, se interpreta respecto al directorio donde está el HTML; si es *absoluto* se interpreta en sentido estricto y puede ser cualquier directorio de la Internet.

7.4.2 Archivos JAR (Java Archives)

Si un ***applet*** consta de varias clases, cada fichero ****.class*** requiere una conexión con el servidor de Web (servidor de protocolo HTTP), lo cual puede requerir algunos segundos. En este caso es conveniente agrupar todos los ficheros en un archivo único, que se puede comprimir y cargar con una sola conexión HTTP.

Los archivos JAR están basados en los archivos ZIP y pueden crearse con el programa ***jar*** que viene con el JDK. Por ejemplo:

```
jar cvf myFile.jar *.class *.gif
```

crea un fichero llamado ***myFile.jar*** que contiene todos los ficheros ****.class*** y ****.gif*** del directorio actual. Si las clases pertenecieran a un package llamado ***es.ceit.infor2*** se utilizaría el comando:

```
jar cvf myFile.jar es\ceit\infor2\*.class *.gif
```

7.5 COMUNICACIÓN DEL APPLET CON EL BROWSER

La comunicación entre el applet y el browser en el que se está ejecutando se puede controlar mediante la interface ***AppletContext*** (package ***java.applet***). ***AppletContext*** es una interface implementada por el browser, cuyos métodos pueden ser utilizados por el ***applet*** para obtener información y realizar ciertas operaciones, como por ejemplo sacar ***mensajes breves*** en la ***barra de estado*** del browser. Hay que tener en cuenta que la barra de estado es compartida por el browser y las ***applets***, lo que tiene el peligro de que el mensaje sea rápidamente sobre-escrito por el browser u otras ***applets*** y que el usuario no llegue a enterarse del mensaje.

Los mensajes breves a la barra de estado se producen con el método ***showStatus()***, como por ejemplo,

```
getAppletContext().showStatus("Cargado desde el fichero " + filename);
```

Los mensajes más importantes se deben dirigir a la ***salida estándar*** o a la ***salida de errores***, que en ***Netscape Navigator*** es la ***Java Console*** (la cual se hace visible desde el menú ***Options*** en ***Navigator 3.0***, desde el menú ***Communicator*** en ***Navigator 4.0**** y desde ***Communicator/Tools*** en ***Navigator 4.5***). Estos mensajes se pueden enviar con las sentencias:

```
System.out.print();
System.out.println();
System.error.print();
System.error.println();
```

Para mostrar documentos HTML en una ventana del browser se pueden utilizar los métodos siguientes:

- ***showDocument(URL miUrl, [String target])***, que muestra un documento HTML en el *frame* del browser indicado por *target* (*name*, *_top*, *_parent*, *_blank*, *_self*).
- ***showDocument(URL miUrl)***, que muestra un documento HTML en la ventana actual del browser.

Un ***applet*** puede conseguir información de otras ***applets*** que están corriendo en la misma página del browser, enviarles mensajes y ejecutar sus métodos. El mensaje se envía invocando los métodos del otro ***applet*** con los argumentos apropiados.

Algunos browsers exigen, para que las ***applets*** se puedan comunicar, que las ***applets*** provengan del mismo browser o incluso del mismo directorio (que tengan el mismo *codebase*).

Por ejemplo, para obtener información de otras applets se pueden utilizar los métodos:

- ***getApplet(String name)***, que devuelve el ***applet*** llamada ***name*** (o ***null*** si no la encuentra). El nombre del ***applet*** se pone con el atributo opcional NAME o con el parámetro NAME.
- ***getApplets()***, que devuelve una *enumeración* con todas las ***applets*** de la página.

Para poder utilizar todos los métodos de un applet que se está ejecutando en la misma página HTML (y no sólo los métodos comunes heredados de ***Applet***), debe hacerse un ***cast*** del objeto de la clase ***Applet*** que se obtiene como valor de retorno de ***getApplet()*** a la clase concreta del ***applet***.

Para que pueda haber ***respuesta*** (es decir, comunicación en los dos sentidos), el primer applet que envía un mensaje debe enviar una referencia a sí misma por medio del argumento ***this***.

7.6 SONIDOS EN APPLETS

La clase ***Applet*** y la interface ***AudioClips*** permiten utilizar sonidos en applets. La Tabla 7.1 muestra algunos métodos interesantes al respecto.

Métodos de Applet	**Función que realizan**
public AudioClip getAudioClip(URL url)	Devuelve el objeto especificado por url, que implementa la interface AudioClip
public AudioClip getAudioClip(URL url, String name)	Devuelve el objeto especificado por url (dirección base) y name (dirección relativa)
play(URL url), play(URL url, String name)	Hace que suene el AudioClip correspondiente a la dirección especificada
Métodos de la interface AudioClip (en java.applet)	**Función que realizan**
void loop()	Ejecuta el sonido repetidamente
void play()	Ejecuta el sonido una sola vez
void stop()	Detiene el sonido

Tabla 7.1. Métodos de Applet y de AudioClip relacionados con sonidos.

Respecto a la carga de sonidos, por lo general es mejor cargar los sonidos en un ***thread*** distinto (creado en el método ***init()***) que en el propio método ***init()***, que tardaría en devolver el control y permitir al usuario empezar a interaccionar con el ***applet***.

Si el sonido no ha terminado de cargarse (en la ***thread*** especial para ello) y el usuario interacciona con el ***applet*** para ejecutarlo, el ***applet*** puede darle un aviso de que no se ha terminado de cargar.

7.7 IMÁGENES EN APPLETS

Las ***applets*** admiten los formatos JPEG y GIF para representar imágenes a partir de ficheros localizados en el servidor. Estas imágenes se pueden cargar con el método ***getImage()*** de la clase ***Applet***, que puede tener las formas siguientes:

```
public Image getImage(URL url)
public Image getImage(URL url, String name)
```

Estos métodos devuelven el control inmediatamente. Las imágenes de cargan cuando se da la orden de dibujar las imágenes en la pantalla. El dibujo se realiza entonces de forma incremental, a medida que el contenido va llegando.

Para dibujar imágenes se utiliza el método ***drawImage()*** de la clase ***Graphics***, que tiene las formas siguientes:

```
public abstract boolean drawImage(Image img, int x, int y,
                                  Color bgcolor, ImageObserver observer)

public abstract boolean drawImage(Image img, int x, int y, int width, int height,
                                  Color bgcolor, ImageObserver observer)
```

El primero de ellos dibuja la imagen con su tamaño natural, mientras que el segundo realiza un cambio en la escala de la imagen.

Los métodos ***drawImage()*** van dibujando la parte de la imagen que ha llegado, con su tamaño, a partir de las coordenadas (x, y) indicadas, utilizando ***bgcolor*** para los pixels transparentes.

Estos métodos devuelven el control inmediatamente, aunque la imagen no esté del todo cargada. En este caso devuelve ***false***. En cuanto se carga una parte adicional de la imagen, el proceso que realiza el dibujo avisa al ***ImageObserver*** especificado. ***ImageObserver*** es una interface implementada por ***Applet*** que permite seguir el proceso de carga de una imagen.

7.8 OBTENCIÓN DE LAS PROPIEDADES DEL SISTEMA

Un ***applet*** puede obtener información del sistema o del entorno en el que se ejecuta. Sólo algunas propiedades del sistema son accesibles. Para acceder a las propiedades del sistema se utiliza un método ***static*** de la clase ***System***:

```
String salida = System.getProperty("file.separator");
```

Los nombres y significados de las propiedades del sistema accesibles son las siguientes:

"file.separator"	Separador de directorios (por ejemplo, "/" o "\")
"java.class.version"	Número de version de las clases de ***Java***
"java.vendor"	Nombre específico del vendedor de Java
"java.vendor.url"	URL del vendedor de Java
"java.version"	Número de versión Java
"line.separator"	Separador de líneas
"os.arch"	Arquitectura del sistema operativo
"os.name"	Nombre del sistema operativo
"path.separator"	Separador en la variable Path (por ejemplo, ":"

No se puede acceder a las siguientes propiedades del sistema: "java.class.path", "java.home", "user.dir", "user.home", "user.name".

7.9 UTILIZACIÓN DE THREADS EN APPLETS

Un ***applet*** puede ejecutarse con varias ***threads***, y en muchas ocasiones será necesario o conveniente hacerlo así. Hay que tener en cuenta que un ***applet*** se ejecuta siempre en un browser (o en la aplicación ***appletviewer***).

Así, las ***threads*** en las que se ejecutan los métodos mayores -***init()***, ***start()***, ***stop()*** y ***destroy()***- dependen del browser o del entorno de ejecución. Los métodos gráficos -***paint()***, ***update()*** y ***repaint()***- se ejecutan siempre desde una ***thread*** especial del AWT.

Algunos browsers dedican un ***thread*** para cada ***applet*** en una misma página; otros crean un grupo de ***threads*** para cada ***applet*** (para poderlas matar al mismo tiempo, por ejemplo). En cuelquier caso se garantiza que todas las ***threads*** creadas por los métodos mayores pertenecen al mismo grupo.

Se deben introducir ***threads*** en ***applets*** siempre que haya tareas que consuman mucho tiempo (cargar una imagen o un sonido, hacer una conexión a Internet, …). Si estas tareas pesadas se ponen en el método ***init()*** bloquean cualquier actividad del ***applet*** o incluso de la página HTML hasta que se completan. Las tareas pesadas pueden ser de dos tipos:

- Las que sólo se hacen una vez.
- Las que se repiten muchas veces.

Un ejemplo de tarea que se repite muchas veces puede ser una animación. En este caso, la tarea repetitiva se pone dentro de un bucle ***while*** o ***do…while***, dentro del ***thread***. El ***thread*** se debería crear dentro del método ***start()*** del ***applet*** y destruirse en ***stop()***. De este modo, cuando el ***applet*** no está visible se dejan de consumir recursos.

Al crear el ***thread*** en el método ***start()*** se pasa una referencia al ***applet*** con la palabra ***this***, que se refiere al ***applet***. El applet deberá implementar la interface ***Runnable***, y por tanto debe definir el método ***run()***, que es el centro del ***Thread*** (ver Apartado 6.1.2, en la página 117).

Un ejemplo de tarea que se realiza una sola vez es la carga de imágenes ****.gif*** o ****.jpeg***, que ya se realiza automáticamente en un ***thread*** especial.

Sin embargo, los sonidos no se cargan en ***threads*** especiales de forma automática; los debe crear el programador para cargarlos en "background". Este es un caso típico de programa ***producer-consumer***: el ***thread*** es el *producer* y el ***applet*** el *consumer*. Las ***threads*** deben estar ***sincronizadas***, para lo que se utilizan los métodos ***wait()*** y ***notifyAll()***.

A continuación se presenta un ejemplo de ***thread*** con tarea repetitiva:

```
public void start() {
    if (repetitiveThread == null) {
        repetitiveThread = new Thread(this); // se crea un nuevo thread
    }
    repetitiveThread.start(); // se arranca el thread creado: start() llama a run()
}

public void stop() {
    repetitiveThread = null; // para parar la ejecución del thread
}


public void run() {
    ...
    while (Thread.currentThread() == repetitiveThread) {
        ... // realizar la tarea repetitiva.
    }
}
```

El método ***run()*** se detendrá en cuanto se ejecute el método ***stop()***, porque la referencia al ***thread*** está a ***null***.

7.10 APPLETS QUE TAMBIÉN SON APLICACIONES

Es muy interesante desarrollar ***aplicaciones*** que pueden funcionar también como ***applets*** y viceversa. En concreto, para hacer que un ***applet*** pueda ejecutarse como ***aplicación*** pueden seguirse las siguientes instrucciones:

1. Se añade un método ***main()*** a la clase ***MiApplet*** (que deriva de ***Applet***)
2. El método ***main()*** debe crear un objeto de la clase ***MiApplet*** e introducirlo en un ***Frame***.
3. El método ***main()*** debe también ocuparse de hacer lo que haría el browser, es decir, llamar a los métodos ***init()*** y ***start()*** de la clase ***MiApplet***.
4. Se puede añadir también una ***static inner class*** que derive de ***WindowAdapter*** y que gestione el evento de cerrar la ventana de la aplicación definiendo el método ***windowClosing()***. Este método llama al método ***System.exit(0)***. Según como sea el ***applet***, el método ***windowClosing()*** previamente deberá también llamar a los métodos ***MiApplet.stop()*** y ***MiApplet.destroy()***, cosa que para las ***applets*** se encarga de hacer el browser. En este caso conviene que el objeto de ***MiApplet*** creado por ***main()*** sea ***static***, en lugar de una variable local.

A continuación se presenta un ejemplo:

```
public class MiApplet extends Applet {
    ...
    public void init() {...}
    ...

    // clase para cerrar la aplicación
    static class WL extends WindowsAdapter {
        public void windowClosing(WindowEvent e) {
            MiApplet.stop();
            MiApplet.destroy();
            System.exit(0);
        }
    } // fin de WindowAdapter

    // programa principal
    public static void main(String[] args) {
        static MiApplet unApplet = new MiApplet();
        Frame unFrame = new Frame("MiApplet");
        unFrame.addWindowListener(new WL());
        unFrame.add(unapplet, BorderLayout.CENTER);
        unFrame.setSize(400,400);
        unApplet.init();
        unApplet.start();
        unFrame.setVisible(true);
    }

} // fin de la clase MiApplet
```