PROGRAMACION CONCURRENTE Y DISTRIBUIDA

III.1 Concurrencia con Java: Thread Java

J.M. Drake
L. Barros

Notas:

Concurrencia en Java.

- Java posibilita la programación concurrente a través de threads.
- Los threads son procesos ligeros, con línea de flujo de control propia pero que comparte el espacio de direcciones del programa.
  - Los threads hacen posible la ejecución concurrente de código.
  - Los cambios de contexto son menos costosos en tiempo de ejecución.

Java es un lenguaje que permite formular programas concurrentes. Esto es, existen en Java sentencias que permiten construir un programa con múltiples líneas de flujo de control (threads) que se ejecutan concurrentemente si la plataforma es multiprocesadora o que ejecutan de forma entralazada sus sentencias dando apariencia de concurrencia si la plataforma es monoprocesadora.

La unidad básica de concurrencia en Java es el Thread, que representa un proceso ligero que comparte el mismo espacio de variables con los restantes threads del mismo programa. Dado que los Threads de un programa Java son todos ellos internos al programa, no se necesita la protección de seguridad de disponer de un único espacio de variables propio de cada threads, y al tener un espacio de memoria único se aligera y facilita los cambios de contexto entre threads.

Como se ha visto en los capítulos previos del curso la concurrencia proporciona muchas ventajas de programación, tanto en la gestión de entradas y salidas, de dispositivos externos o de las interacciones con el usuario, como para hacer que la estructura del programa sea mas próxima al dominio del problema al facilitar la programación orientada a objetos y con objetos activos.

En contrapartida la programación concurrente requiere abordar nuevos situaciones sobre sincronización entre threads y acceso de forma segura a los objetos compartidos, por lo que el lenguaje Java debe proporcionar sentencias y recursos que permitan abordarlas.

Class Thread

- Un thread se crea en Java instanciando un objeto de la **clase Thread**.
- El **código que ejecuta** un thread está definido por el método **run()** que tiene todo objeto que sea instancia de la clases Thread.
- La ejecución del **thread se inicia** cuando sobre el objeto Thread se ejecuta el método **start().**
- De forma natural, un **thread termina** cuando en run() se alcanza una sentencia **return** o el final del método. (Existen otras formas de terminación forzada)

Notas:

Para declarar un thread en un programa Java se debe instanciar un objeto que sea una extensión (extends) la clase "java.lang.Thread". Por cada objeto de con estas características que se declare en un programa java, se crea un nuevo thread con una línea de flujo de control propia que se ejecutará concurrentemente con el programa principal y con los restantes threads.

El código que ejecuta un thread es el expresado através de su método run(), que es heredado de la clase Thread. A fin de definir la actividad que se quiere que ejecute el nuevo thread es necesario sobreescribir el método run() original con un código que establezca la actividad deseada.

La ejecución del thread creado con la instanciación de un objeto derivado de Threads comienza cuando se invoca en el programa principal (o en otro thread activo) el método start del objeto.

Existen muchas formas de concluir un threads. La natural es cuando en el método run() que define su actividad, se alcanza alguna sentencia end. Más adelante se analizarán otros métodos heredados de Thread que permiten abortar o elevar excepciones en el thread, y que pueden conducir a su finalización.

Constructores de la clase Thread.

- **Thread**()
- **Thread**(Runnable *threadOb*)
- **Thread**(Runnable *threadOb*, String *threadName*)
- **Thread**(String *threadName)*
- **Thread**(ThreadGroup *groupOb***,** Runnable *threadOb)*
- **Thread**(ThreadGroup *groupOb***,** Runnable *threadOb*, String *threadName*);
- **Thread**(ThreadGroup *groupOb,* String *threadName*)

Notas:

Existen varios constructores de la clase Thread (y transferido por herencia a todas su extensiones). Desde el punto de vista estructural existen dos variantes básicas:

- Las que requieren que el código del método run() se especifique explícitamente en la declaración de la clase.

Por ejemplo: **Thread**(String *threadName)*

- Las que requieren un parámetro de inicializanción que implemente la interfaz Runnable.

Por ejemplo: **Thread**(Runnable *threadOb*)

Los restantes constructores resultan de si se asigna un nombre la threads, y que solo afecta para inicializar ese atributo en la instancia del objeto, y pueda utilizarse para que en fases de verificación cada thread pueda autoidentificarse, o de si se le crea dentro de un ThreadGroup, lo cual limita su accesibilidad y su capacidad de interacción con threads que han sido creados en otros ThreadGroup.

Creación de un thread por herencia.

```
public class PingPong extends Thread{
  private String word;    // Lo que va a escribir.
  private int delay;      // Tiempo entre escrituras
  public PingPong(String queDecir,int cadaCuantosMs){
      word = queDecir; delay = cadaCuantosMs;  };
   public void run(){  //Se sobrescribe run() de Thread
      while(true){
            System.out.print(word + " ");
            try{
                sleep(delay);
            } catch(InterruptedException e){ return; }
      }
  }
}
```

```
  public static void main(String[] args){
            // Declaración de 2 threads
     PingPong t1 =new PingPong("PING",33);
     PingPong t2= new PingPong("PONG",10);
            // Activación
     t1.start();
     t2.start();
            // Espera 2 segundos
     try{ sleep(5000);
     }catch (InterruptedException e){};
            // Finaliza la ejecución de los threads
     t1.stop();
     t2.stop();
  }
}
```


La clase java PingPong es una extensión de la clase Threads, por lo que cada una de sus instancias (por ejemplo t1 y t2 representan un nuevo thread java.

El metodo run() es sobreescrito en la clase PingPong, y establece el código que se ejecutará en cada instancia de la clase. En el ejemplo consiste en un bucle indefinido de escribir el valor del atributo word, y de suspenderse durante los milisegundos expresador en el atributo delay.

La sentencia try .. catch tiene que ser utilizada porque el método sleep puede elevar una excepción del tipo InterruptException.

Con las sentencias que declaran e instancian los dos objetos t1 y t2 de la clase PingPong se han creado dos threads. La creación supone que a los thread se les ha dotado de los recursos que requieren, pero aun están inactivos (esto es no se está ejecutando las sentencias de su método run()).

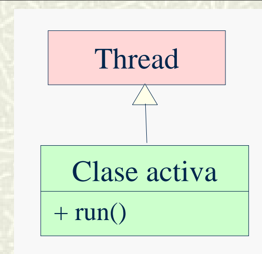
Con las sentencias que invocan los métodos t1.start() y t2.start() de los objetos de la clase PingPong, se inicia en la ejecución del código de los respectivos procedimiento run().

En este caso (para hacer el ejemplo sencillo) los threads no finalizan nunca (observese el bucle while(true) de los métodos run()). Se acabarán desde el entorno abortando el programa principal y todo lo que depende de él.

Interface Runnable

Definición de la interface Runnable:

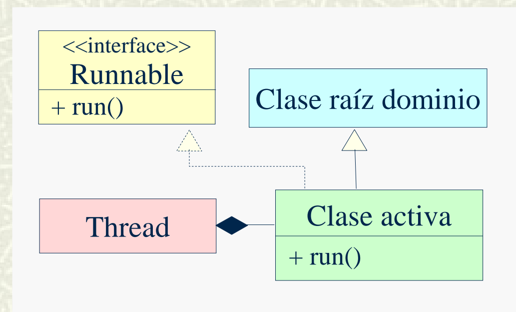
```
public interface Runnable {
    public abstract void run();
}
```



Creación de un thread a partir de la interface Runnable:

1)
```
public class MiClase implements Runnable{
    public void run(){ ......}
}
```

2)
```
.....
MiClase  ot = new MiClase();
Thread t1 = new Thread(ot);
t1.start();
.....
```



La segunda posibilidad de crear un thread es a través de la utilización de un objeto que implemente la interface Runnable, y con el que se incorpora el método run() que establece las actividades que va a realizar.

La clase MiClase implementa la interfaz Runnable, y por ello se le requiere que establezca el código del método abstracto run() al que obliga ser la implementación de la interfaz Runnable.

En el programa que declara el nuevo thread, se debe declarar primero el objeto t1 de la clase MiClase, y posteriormente cuando se crea el threads (se instancia el objeto t1 de la clase Thread) se le pasa como parámetro de su constructor.

Este es el procedimiento mas habitual de crear threads en java, ya que permite herencia múltiple, al poder ser la clase Runnable extensión de cualquier otra clase que ya esté definida.

Creación de Thread a través de la interfaz Runnable.

```
public class PingPong implements Runnable {
   private String word;     // Lo que va a escribir.
   private int delay;       // Tiempo entre escrituras
   public PingPong(String queDecir, int cadaCuantosMs){
       word = queDecir; delay = cadaCuantosMs; };

public void run(){
        while(true){
             System.out.print(word + " ");
             try{ sleep(delay);
             }catch(InterruptedException e){return;}
        }
   }

}
```

```
public class PruebaRunnable{
   public static void main(String[] args){
     // Los objetos r1 y r2 definen la funcionalidad.
     // (definen los métodos run())
       PingPong r1 =new PingPong("PING",33);
       PingPong r2= new PingPong("PONG",10);
       // Se crean los threads
        Thread t1 = new Thread(r1);
        Thread t2= new Thread(r2);
        // Se activan los threads
        t1.start();
        t2.start();
   }
}
```

Procodis'09: III.1- Thread Java   José M.Drake   7

Notas:

La clase PingPong implementa la interfaz Runnable, y en consecuencia define el método run() que constituirá el cuerpo de futuros threads.

Aun que no ocurre en el ejemplo, la clase PingPong podría ser una extensión de otra clase, de la que heredaría su funcionalidad y sus recursos.

Cuando instancian los threads t1 y t2 (en este caso directamente de la clase Thread, aunque podría ser de cualquier otra derivada de ella), se le pasa como valor actual del parámetro Runnable threadOb del constructor, los objetos Runnables r1 y r2 de la clase PingPong que fueron previamente instanciados.

La declaración de los objetos runnables podría ser directa dentro de la declaración de los threads, solo que en ese caso serían anónimos:

```
PingPong t1 = new Thread(new PingPong("ping",33));
PingPong t2 = new Thread(new PingPong("PONG",100));
```

O incluso con los thread también anónimos:

```
new Thread(new PingPong("ping",33));
new Thread(new PingPong("PONG",100));
```

Delegación en una operación costosa en un thread

```
public class threadAgente{
  ....
 public void operacionCostosa(){
   ....
 };

 static public void main(String[] args){
    ....
    // Lanza la ejecución costosa en un thread anónimo concurrente
    new Thread(
              new Runnable(){
                 public void  run(){ operacionCostosa();}
              }
         ).start();
   //main ejecuta concurrentemente otra tarea de interés
   ....
    };
}
```


Notas:

Clase activa con thread auto lanzado

```
public class SelfRun implements Runnable{
  private Thread internalThread;
  private boolean noStopRequired;
  public SelfRun(){
    System.out.println("Comienza ejecución");
    noStopRequired=true;
    internalThread=new Thread(this);
    internalThread.start(); }
  public void run(){
     while(noStopRequired){
        System.out.println("En ejecución");
        try{ Thread.sleep(500);
        }catch (InterruptedException e){
            Thread.currentThread().interrupt();}
     }
  }
  public void stopRequest(){
  noStopRequired=false;
  internalThread.interrupt();}
```

```
public static void main(String[] args) {
      SelfRun objActivo=new SelfRun();
      // Espera durante 2 segundos
      try{Thread.sleep(2000);
      }catch(InterruptedException e){};
      // Termina el objeto activo
      System.out.println("main invoca stopRequest()");
      objActivo.stopRequest();
   }
}
```

```
Respuesta del programa:
Comienza ejecución
En ejecución
En ejecución
En ejecución
En ejecución
En ejecución
main invoca stopRequest()
```


Notas:

Métodos de la clase Thread: Inicio y finalización.

- void **run**()
  - Contiene el código que ejecuta el thread.
- void **start**()
  - Inicia la ejecución del thread.
- void **stop**()
  - Termina la ejecución del thread (*)
- static Thread **currentThread**()
  - Retorna el objeto Thread que ha ejecutado este método.
- final void **sleep**(long milliseconds) **throws** InterruptedException
  - Suspende la ejecución del thread por el número de milisegundos especificados.

El método stop() es muy inseguro. Si se destruye un thread con recursos tomados estos permanecen tomados y no accesibles a otros componentes de la aplicación. Es un método ya obsoleto que será eliminado en las próximas versiones. Un thread debe terminarse utilizando el método interrupt().

El método sleep puede lanzar la excepción InterruptedException (lo que ocurre si mientras que el thread está suspendido, otro threads ejecuta el método Interrupt de él). Por esta razón, el método sleep debe ejecutarse siempre dentro de una sentencia try .. catch:

```
try{
    ....
    sleep(500);
    ....
}
catch (InterruptedException e {...};
```

Ejemplo de uso del método currentThread().

```
public class Cliente extends Thread{

  public void run(){
     Recurso.uso();
     try {
        Thread.sleep(2000);
     } catch(InterruptedException e){};
  };
}

public class Recurso{
     static synchronized void uso(){
         Thread t = Thread.currentThread();
         System.out.println("Soy "+t.getName());
     }
}
```

```
  public static void main(String[]
     args){
     Cliente juan= new Cliente();
     juan.setName("Juan López");
     Cliente ines= new Cliente();
     ines.setName("Inés García");
     juan.start();
     ines.start();
  }
}
```

Procodis'09: III.1- Thread Java José M.Drake 11

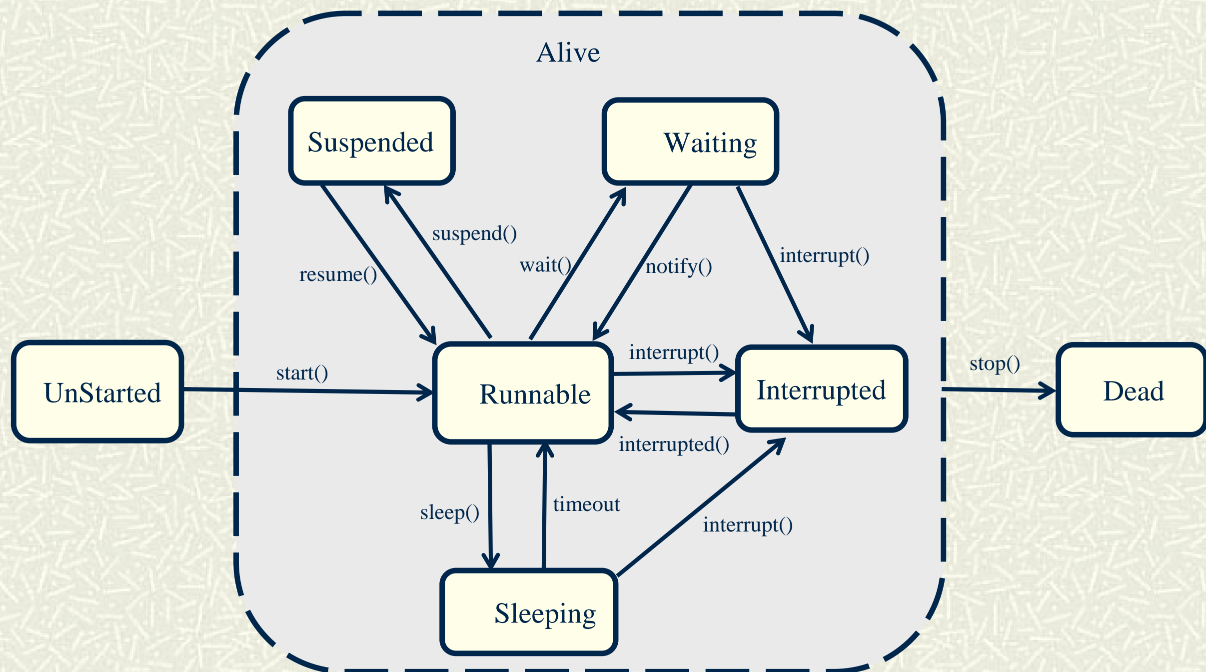
En el ejemplo se instancian varios threads (juan e ines) del tipo Cliente que es una extensión de la clase Thread.

En la definición de su actividad (método run()), invocan el método uso de un objeto construido con la clase estática Recurso.

Dentro del objeto Recurso cada thread se autoidentifica a través del método currentThread,(y en este caso imprime su identificador por defecto: nombre,prioridad, Grupos, etc.). Con ello, se puede escribir en un recurso compartido por muchos threads sentencias que se particularicen en función de que thread lo está visitando.

El nombre de los thread se establece a través del método setName, se podría haber hecho de forma equivalente a través del contructor:

```
Cliente juan = new Cliente("Juan López");
Cliente ines = new Cliente("Ines García");
```



Estado "UnStarted": El thread ha sido instanciado pero no se arrancado su ejecución.

Estado "Alive": El código del método run() se está ejecutando.

Subestado "Runnable": La ejecución del thread esta siendo planificada por el procesador.

Subestado "Interrupted": La ejecución del thread está siendo planificada por el procesador, pero ero se ha invocado el método Interrupt sobre el thread.

Subestado "Suspended": El thread está supendido indefinidamente (porque sobre él se ha invocado el método suspend(). El thread no está siendo planificado.

Subestado "Sleeping": El thread está supendido temporalmente (durante el número de ms establecido por el argumento del método sleep() que lo ha dormido).

Subestado "Waiting": El thread está supendido indefinidamente (sobre la variable de condición del objeto thread por haber invocado el mismo el método wait()).

Estado "Dead": El thread ha finalizado. Un thread finaliza por uno de los tres siguientes modos:

- El método run() ejecuta la sentencia return o alcanza su final.
- Se invoca desde un thread activo el método stop().
- No se recupera la excepción InterruptedException estando en Waiting o Sleeping.

Métodos de la clase Thread: Control de finalización.

- final boolean **isAlive**()
  - Retorna true si el thread se encuentra en el estado Alive (en alguno de sus subestados), esto es, ya ha comenzado y aun no ha terminado.
- final void **join**() throws InterruptedException
  - Suspende el thread que invoca hasta que el thread invocado haya terminado.
- final void **join**(long *milliseconds*) throws InterruptedException
  - Suspende el thread que invoca hasta que el thread invocado haya terminado o hasta que hayan transcurrido los milisegundos.

Notas:

Ejemplo de uso del método isAlive().

```
public class Obrera extends Thread{
  private String resultado = "No calculado";

  public void run(){
     resultado = calcula();
  }

  private String calcula(){
       // Realiza un cálculo largo.
    try {Thread.sleep(10000);
    } catch(InterruptedException e){};
    return "Ya calculado";
   }
 public String getResultado(){
    return resultado;
 }
}
```

```
class Ejemplo_isAlive {
   public static void main(String[] args){
     Obrera agente =new Obrera();
     agente.start();
      // Hace algo durante el cálculo.
     ......
           //Espera que agente haya terminado
     while (agente.isAlive()) Thread.yield();
           // Utiliza el resultado.

    System.out.println(agente.getResultado());
   }
}
```


El programa principal "main" instancia el thread "agente" de la clase "Obrera" (es un thread porque es una extensión de Thread), y delega en él que para que invoque y ejecute Calcula().

Concurrentemente el programa main ejecuta alguna otra actividad. Cuando finaliza se introduce en un lazo de espera activa en el que haciendo uso del método isAlive espera a que la Obrera agente haya concluido.

Luego lee de ella el resultado. Observese que aunque ha concluido se pueden acceder a su variable resultado a través del método getResultado().

ESTE ES UN PROGRAMA MAL DISEÑADO PORQUE REQUIERE UNA ESPERA ACTIVA.

Ejemplo de uso del método join()

```
public class Obrera extends Thread{
  private String resultado = "No calculado";

  public void run(){
     resultado = Calcula();
  }

  String Calcula(){
     // Realiza un cálculo largo.
     try {Thread.sleep(10000);
     }catch(InterruptedException e){};
     return "Ya calculado";
   }

 public String getResultado(){
   return resultado;
 }
}
```

```
class Ejemplo_join {
   public static void main(String[] args){
      Obrera agente = new Obrera();
      agente.start();
      // Hace algo durante el cálculo.

      try {     //Espera a que agente termine
         agente.join();
      }catch (InterruptedException e){};
            // Utiliza el resultado.

      System.out.println(agente.getResultado());
   }
}
```


Este programa tiene la misma funcionalidad que el programa de la página anterior (12).

Sin embargo este programa es mucho mas eficiente, esto es requerirá menos tiempo de CPU para ser ejecutado, ya que la espera ahora es suspendida (Waiting). El programa principal main invoca el método join sobre el thread Obrera agente. Main pasa a estado Waiting y espera suspendido allí hasta que tras la finalización del thread agente, lo retorna al estado Runnable.

La invocación del método join hay que invocarlo dentro de una sentencia try .. catch, ya que si sobre main se invoca el método interrupt, se elevara la excepción InterruptedException.

Métodos de la clase Thread: Interrupción.

- void **interrupt**()
  - El thread pasa a estado Interrupted. Si está en los estados Waiting, Joining o Sleeping termina y lanza la excepción InterruptedException. Si está en estado Runnable, continua ejecutándose aunque cambia su estado a Interrupted.
- static boolean **interrupted**()
  - Procedimiento estático. Retorna true si el thread que lo invoca se encuentra en estado Interrupted. Si estuviese en el estado Interrupted se pasa al estado Runnable.
- boolean **isInterrupted**()
  - Retorna true si el objeto thread en que se invoca se encontrase en el estado Interrupted. La ejecución de este método no cambia el estado del thread.

Notas:

Uso del método isInterrupted().

```
public class InterruptCheck {

public static void main(String[] args) {
   Thread t=Thread.currentThread();
   System.out.println("A:t.isInterrupted()="+ t.isInterrupted());
    t.interrupt();
    System.out.println("B:t.isInterrupted()="+ t.isInterrupted());
    System.out.println("C:t.isInterrupted()="+ t.isInterrupted());
    try{
        Thread.sleep(2000);
        System.out.println("No ha sido interrumpida");
    }catch(InterruptedException e){
        System.out.println("Si ha sido interrumpida");
    }
     System.out.println("D:t.isInterrupted()="+ t.isInterrupted());
}
```

- **No cambia**

Respuesta del programa:

```
A:t.isInterrupted()=false
B:t.isInterrupted()=true
C:t.isInterrupted()=true
No ha sido interrumpida
D:t.isInterrupted()=false
```


Notas:

Uso del metodo estático Thread.interrupted()

```
public class InterruptReset {
  public static void main(String[] args) {
    Thread t=Thread.currentThread();
    System.out.println("A: Thread.interrupted()="+Thread.interrupted());
    t.interrupt();
    System.out.println("B: Thread.interrupted()="+Thread.interrupted());
    System.out.println("C: Thread.interrupted()="+Thread.interrupted());
    try{
      Thread.sleep(2000);
      System.out.println("No ha sido interrumpida");
    }catch(InterruptedException e){
            System.out.println("Si ha sido interrumpida");
    }
    System.out.println("D: t.isInterrupted()="+t.isInterrupted());
  }
}
```

•cambia

Respuesta del programa:

```
A: Thread.interrupted()=false
B: Thread.interrupted()=true
C: Thread.interrupted()=false
No ha sido interrumpida
D: t.isInterrupted()=false
```


Notas:

Ejemplo de finalización por InterruptedException.

```
public class MiThread extends Thread{

  public void run(){
    while (true) {
      System.out.println("Ejecuto");
      try{
        Thread.sleep(100);
      }catch (InterruptedException e){
         System.out.println("Termino en sleep");
      };
    };
    return;
  }
}
```

```
public class Ejemplo_Fin_por_Interrupt {

  public static void main(String[] args){
    Thread elThread= new MiThread();
    elThread.start();
     try{
       Thread.sleep(1000);
     }catch (InterruptedException e){};
     elThread.interrupt();
   }
}
```


Notas:

El thread “elThread” ejecuta un bucle periódico que escribe “Ejecuto” y luego se suspende durante 100 ms.

El programa principal main, espera durante un tiempo (1000 ms) y luego invoca el método interrupt() del thread “elThread”.

Cuando se invoca el método interrupt() elTthread puede estar en uno de dos estados:

- Si está en el estado Runnable, pasa al estado Interrupted hasta que se invoque el método sleep() y pase al estado Sleeping, instante en el que se eleva la excepción InterruptedException en elThread, y tras escribir el comentario finaliza elThread.
- Si está en el estado Sleeping, se eleva de forma inmediata la excepción InterruptedException en elThread, y tras escribir el comentario finaliza elThread.

Ejemplo de uso de interrupt() e interrupted()

```
public class MiThread extends Thread{

   public void run(){
     while (!Thread.interrupted()) {
             System.out.println("Ejecuto");
     };
     System.out.println("Termino");
     return;
   }
}
```

```
public class Ejemplo_Fin_por_Interrupt {

   public static void main(String[] args){
     Thread elThread= new MiThread();
     elThread.start();
      try{
        Thread.sleep(1000);
      }catch (InterruptedException e){};
      elThread.interrupt();
   }
}
```


En este caso, elThread permanece permanentemente es el estado Runnable, ejecutando la escritura del texto “Ejecuto” en un bucle indefinido que perdura mientas que el resultado de invocar interrupted() sea false.

Cuando tras 1000 ms main invoque el método interrupted() sobre elThread, este pasa a estado Interrupted, y en la siguiente ejecución de la sentencia while sale del bucle, y tras escribir el texto de salida, ejecuta return y concluye.

Interrupción durante sleep

```
public class SleepInterrupt implements Runnable {
  public void run(){
    try{
    System.out.println("in run(): me duermo 20 s");
    Thread.sleep(20000);
     System.out.println("in run(): me despierto");
     }catch (InterruptedException e){
          System.out.println(
                     "in run(): interrumpida en sleep");
          return;
    }
    System.out.println("in run(): fin normal");
  }
  ....
```

```
 ...
 public static void main(String[] args) {
   SleepInterrupt si=new SleepInterrupt();
   Thread t=new Thread(si);
    t.start();
    try{
       Thread.sleep(1000);
     }catch (InterruptedException e){};
    System.out.println("in main(): Intterupo a t");
    t.interrupt();
    System.out.println("in main(): termina");
  }
}
```

```
Respuesta del programa:
in run(): me duermo 20 s
in main(): Intterupo a t
in run():  interrumpida en sleep
in main(): termina
```


Notas:

Interrupción pendiente y sleep()

```
public class PendingInterrupt {
  public static void main(String[] args) {
      if (args.length>0){ Thread.currentThread().interrupt();}
      long tiempoInicial=System.currentTimeMillis();
      try{
         Thread.sleep(2000);
          System.out.println("No es interrumpida");
      }catch (InterruptedException e){
      System.out.println("Es interrumpida");
      }
      System.out.println("Tiempo gastado: "+
      (System.currentTimeMillis()-tiempoInicial));
  }
}
```

Respuesta a java PendingInterrupt:

```
No es interrumpida
Tiempo gastado: 2000
```

Respuesta a java PendingInterrupt yes:

```
Es interrumpida
Tiempo gastado: 0
```


Notas:

Métodos suspend() y resume() de la clase Thread.

- El método **suspend**() permiten parar reversiblemente la ejecución de un Thread. Se reanuda cuando sobre él se ejecute el método **resume**().
- Ejecutar suspend sobre un thread suspendido o ejecutar resume() sobre un thread no suspendido no tiene ningún efecto.
  - public final void **suspend**()
    - Suspende la ejecución del thread. Los objetos sobre los que tenía derecho de acceso quedan cerrados para el acceso de otros threads hasta que tras que le sea aplicado resume(), los libere.
  - public final void **resume**()
    - Reanuda el thread si ya estuviera suspendido.

Los metodos suspend() y resume() será eliminado en futuras versiones de java, al igual que el método stop() ya que son muy proclives a provocar bloqueos.

Ejemplo de aplicación de métodos suspend() y resume()

- El procedimiento ConfirmaCancelacion() suspende la actividad del thread que se pasa como el argumento elThread, mientras se confirma su abandono definitivo.

```
public void ConfirmaCancelacion(Thread elThread) {
  elThread.suspend();
  if (DebeCancelarse("Realmente se cancela?"))
    elThread.interrupt();
  else
    elThread.resume();
}
```


El ejemplo consiste en el procedimiento ConfirmaCancelacion para confirmar la suspensión definitiva de un thread que se pasa como el argumento elThread.

Inicialmente se invoca sobre él el método suspend() para suspenderlo hasta que el operador haya establecido su destino.

Posteriormente se del operador que confirme o cancele la suspención. En el primer caso, se invoca el método interrupt() para provocar la cancelación ordenada del thread, y en el segundo caso se invoca el método resume() para que continúe desde donde se suspendió.

Métodos de la clase Thread: Control de planificación.

- Constantes
  - MAX_PRIORITY //Máxima prioridad asignable al thread.
  - MIN_PRIORITY //Mínima prioridad asignable al thread.
  - NORM_PRIORITY //Prioridad por defecto que se asigna.

- Métodos
  - final void **setPriority**(int *priority*)
    - Establece la prioridad de Thread.
  - final int **getPriority**()
    - Retorna la prioridad que tiene asignada el thread.
  - static void **yield**()
    - Se invoca el planificador para que se ejecute el thread en espera que corresponda.

La planificación de los thread está pobremente establecida en java, ya que no es un lenguaje de tiempo real, y la mayoría de las Maquinas virtuales java no son sensibles a la prioridad de los threads.

De acuerdo con la especificación original la planificación debería realizarse con una política expulsora ("pre-emptive") y basada en prioridades.

Cada clase especializada de Thread, ofrece tres atributos constantes que pueden ser leídos y que expresan los límites de prioridades (MAX_PRIORITY,MIN_PRIORITY) que pueden asignarse a cada thread de la clase, y el valor de prioridad por defecto (NORM_PRIORITY).

La norma de planificación establece que el planificador nunca planifica un thread de entre los que se encuentran en el estado Runnable o Interrupted si hay otro thread en los citados estados con mayor prioridad asignada que él.

Manejo de prioridades

```
public class SimplePriorities extends Thread {
  private int countDown = 5;
  private volatile double d = 0; // No optimization
  public SimplePriorities(int priority) {
    setPriority(priority);
    start();
  }
  public String toString() {
    return super.toString() + ": " + countDown;
  }
  public void run() {
    while(true) {
      // An expensive, interruptable operation:
      for(int i = 1; i < 100000; i++)
        d = d + (Math.PI + Math.E) / (double)i;
      System.out.println(this);
      synchronized(this){
      if(--countDown == 0) return;
      }
    }
  }
```

```
public static void main(String[] args) {
   new SimplePriorities(Thread.MAX_PRIORITY);
   for(int i = 0; i < 5; i++)
     new SimplePriorities(Thread.MIN_PRIORITY);
  }
}
```


Notas:

Métodos de la clase Thread: Debuging y control.

- String **toString**()
  - Retorna un string con el nombre, prioridad y nombre del grupo del Thread.
- final String **getName**()
  - Retorna el nombre del Thread
- final void **setName**(String *threadName*)
  - Establece el nombre del Thread.
- static int **activeCount**()
  - Retorna el número de threads activos en el grupo al que pertenece el thread que invoca.
- final ThreadGroup **getThreadGroup**()
  - Retorna el grupo al que pertenece el thread que invoca.

Son métodos que se utilizan habitualmente en las fase de validación de los programas.

Métodos de la clase Thread: Daemon.

Existen dos tipos de thread "user" y "daemon ". Su diferencia radica en su efecto sobre la finalización de la aplicación a la que pertenecen.

- Una aplicación termina solo si han terminado todos los threads de tipo user.
- Cuando todos los threads de tipos user han terminado los thread de tipo Daemon son terminados con independencia de cual sea su estado y la aplicación es finalizada.

Metodos relativos al tipo de Thread.

- final void **setDaemon**(boolean *state*)
  - Convierte un thread en daemon. Solo es aplicable después de haber creado el thread y antes de que se haya arrancado (start).
- final boolean **isDaemon**()
  - Retorna true si el thread es de tipo daemon.

Ejemplo de thread Daemond

```
public class DaemondThread implements Runnable{
  public void run(){
    System.out.println("Comienza run()");
    try{
      while (true){
        try{Thread.sleep(500);
        }catch (InterruptedException e){};
        System.out.println("run() ha despertado");
      }
    }finally{
       System.out.println("Termina run()");
    }
  }
```

```
public static void main(String[] args){
    System.out.println("Comienza main()");
    Thread t=new Thread(new DaemondThread());
    t.setDaemon(true);
    t.start();
   try{Thread.sleep(2000);
   }catch (InterruptedException e){};
   System.out.println("Termina main()");
  }
}
```

Respuesta de la ejecución de main()

```
Comienza main()
Comienza run()
run() ha despertado
run() ha despertado
run() ha despertado
run() ha despertado
Termina main()
run() ha despertado
```

Procodis'09: III.1- Thread Java José M.Drake 29

Notas:

Los thread que han sido cualificado como daemon terminan de una forma diferente. Cuando la VM detecta que sólo permanecen en ejecución thread Daemon termina su ejecución.

Los daemon thread son utilizados para ejecutar tareas auxiliares de otros thread normales, cuando estos han terminado aquellos no tienen función y finalizan.

En el ejemplo es interesante comprobar que cuando finaliza main, y la VM descuble que sólo queda el thread t , este finaliza de forma abrupta y no sale normalmente por la rama finally.

Ejemplo de thread User

```
public class DaemondThread implements Runnable{
 public void run(){
   System.out.println("Comienza run()");
   try{
     while (true){
        try{Thread.sleep(500);
        }catch (InteruptedException e){};
        System.out.println("run() ha despertado");
     }
   }finally{
       System.out.println("Termina run()");
   }
 }
```

```
public static void main(String[] args){
    System.out.println("Comienza main()");
   Thread t=new Thread(new DaemonThread());
   t.setDaemon(true);
   t.start();
  try{Thread.sleep(2000);
  }catch (InterruptedException e){};
  System.out.println("Termina main()");
 }
}
```

Respuesta de la ejecución de main()

```
Comienza main()
Comienza run()
run() ha despertado
run() ha despertado
run() ha despertado
run() ha despertado
Termina main()
run() ha despertado
run() ha despertado
....
```

Procodis'09: III.1- Thread Java José M.Drake 30

Notas:

Los thread que han sido cualificado como daemon terminan de una forma diferente. Cuando la VM detecta que sólo permanecen en ejecución thread Daemon termina su ejecución.

Los daemon thread son utilizados para ejecutar tareas auxiliares de otros thread normales, cuando estos han terminado aquellos no tienen función y finalizan.

En el ejemplo es interesante comprobar que cuando finaliza main, y la VM descuble que sólo queda el thread t , este finaliza de forma abrupta y no sale normalmente por la rama finally.