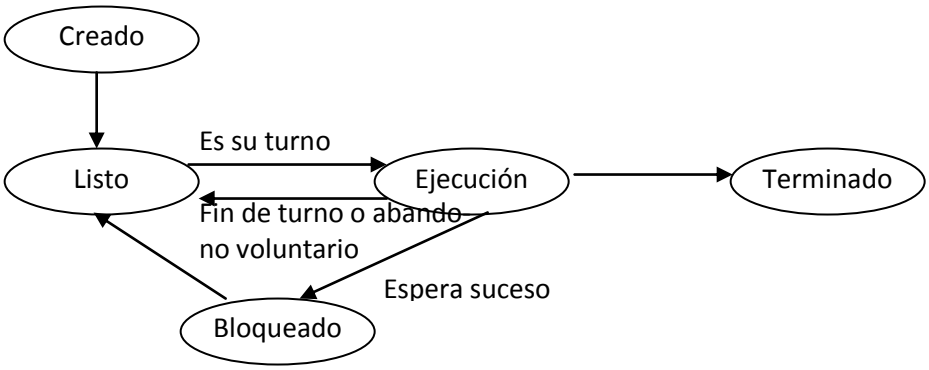
PROCESOS VS. HILOS:

Ciclo de vida de un proceso: es prácticamente un estándar en todos los sistemas operativos. En principio el proceso no existe, es creado, luego pasa a listo (el proceso esta en condiciones de usar la cpu) hasta que se el planificador de procesos scheduler (que suele ser parte de el sistema operativo) le dé la oportunidad de usar el procesador.

Los procesos tienen estado listo hasta que el planificador decide darles tiempo de ejecución pasando al estado en ejecución.



Un proceso puede pasar de ejecución a listo, esta decisión la toma el planificador. Que sigue alguna política para asignar la CPU a los distintos procesos. Una forma que lo realiza es mediante la asignación de un quantum de tiempo, de tal forma que cuando un procesador cumpla su tiempo de permanencia en el procesador, es desalojado pasando nuevamente a estado listo. Puede que un proceso abandone voluntariamente la cpu y pase a listo.

Un proceso puede pasar de estado en ejecución a bloqueado cuando ha de esperar que suceda un evento o suceso. Ejemplo: esperar un operación de entrada salida, la espera de finalización de una tarea por parte de otro proceso, un bloqueo voluntario durante un determinado tiempo, etc. Una vez ocurrido el evento que estaba esperando, el proceso pasa a listo. El acto de cambiar un proceso de estado se denomina **cambio de contexto.**

Proceso en Pascal FC.

Si bien Pascal-FC no trae algunas de las características de Pascal como:

1. Ficheros
2. Punteros
3. Registros variantes
4. No se puede usar la palabra reservada with
5. El tipo set
6. El tipo rango
7. No se puede usar el tipo string, salvo en la instrucción writeln como una constante.

Incorpora otras como:

1. El bucle **repeat…forever**, que define un bucle infinito.
2. La sentencia **null,** si bien no tiene ningún efecto, se utiliza para indicar que no se ejecuta nada.

3. La función random cuyo uso es: i:=random(n). devuelve un número aleatorio entre 0 y abs(n).
4. El programa constituye la única unidad de compilación en Pascal-FC. La estructura de un programa es:

   Program identificador;
   Declaraciones globales
   Begin
   Sentencias
   End.

Donde declaraciones globales pueden ser:

1. Constantes
2. Tipos
3. Variables
4. Procedimientos
5. Funciones
6. Tipos de procesos
7. Procesos
8. Recursos
9. Monitores

La categoría **Sentencias,** está formada por sentencias secuenciales y concurrentes.

Declaración de procesos

En el siguiente ejemplo se muestra la definición de dos procesos y un programa principal que los lanza a ejecución dentro del par **cobegin/coend.** Cada proceso escribe un número dentro de un bucle infinito.

```
Program ejemplo1;
        (* Aquí va la declaración de tipos, constantes y variables globales *)
        (*A continuación la declaración de procesos que componen el programa *)
        Proceso Uno;
        begin
                repeat
                        Writeln(1);
                forever
        end;
        Proceso Dos;
        begin
                repeat
                        Writeln(2);
                forever
        end;
```

```
var
 (* definición de variables globales *)
begin
        writeln('Esto se ejecuta en forma secuencial');
        writeln(' previa a la ejecución concurrente de los procesos');
cobegin
        (* Aquí se activan los procesos y se ejecutan de forma concurrente´*);
        Uno;
        Dos;
coend;
writeln(´Esto se ejecuta de forma secuencial´);
writeln('Despues de la ejecución concurrente de los procesos´)
end.
```

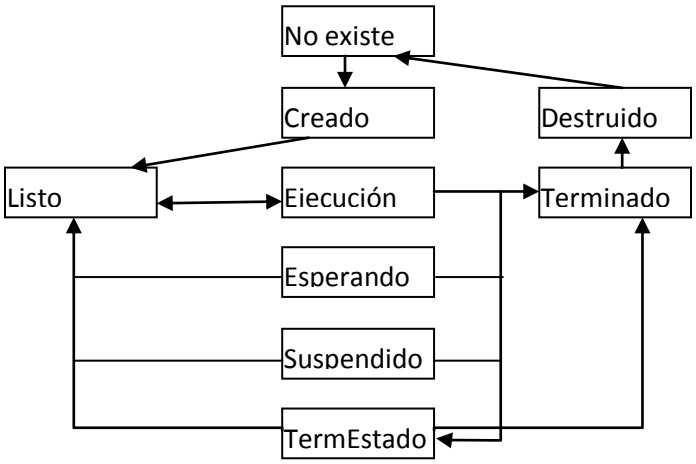
Una posible salida sería:
1112121222112….(indefinidamente)
Se puede observar que ambos procesos son similares, solo cambia el número a imprimir. Pascal-FC permite la creación de tipos de procesos, y luego crear y lanzar procesos pertenecientes a esos tipos. De esta forma, el ejemplo anterior podría quedar como se muestra a continuación. El ejemplo también sirve para ilustrar cómo es posible pasar parámetros a los procesos:

```
Program ejemplo2;
        Process type Proceso(s:integer);
        begin
                repeat
                        Writeln(s);
                forever
        end;

        var
         a,
        b:Proceso; (* creación de procesos *)
        begin
            cobegin
                a(1);
                b(2);
            coend;
end.
```

Estados de un proceso en Pascal-FC.

El compilador genera el código con todas las facilidades de mantenimiento y planificación de procesos, este código (denominado Run-time Support System o RTSS) junto al código generado por el programa usuario, es lanzado para formar un solo programa, que será una tarea del sistema operativo.
El mayor inconveniente es el comportamiento de un programa cuando uno de sus procesos



hace una operación de E/S, otro proceso podría adueñarse del procesador. Sin embargo desde el punto de vista del Sistema Operativo, el programa hace una E/S, así que se bloqueará hasta completar la E/S. el siguiente ejemplo no imprimirá 'hola' mientras el proceso lector esté esperando un dato. Es importante tener en cuenta esto, para el correcto funcionamiento de los programas que haremos con este lenguaje

```
program Problema
process Escritor;
begin
        repeat
          writeln('hola');
        forever
end;
```

```
process Lector;
begin
        repeat
          read(dato)
        forever
end;

begin
        cobegin
          Escritor;
          Lector;
end.
```

Planificación de procesos en Pascal-FC.

Las dos políticas existentes para hacer la planificación de procesos en Pascal-FC son:

1. Ejecutar un proceso hasta que termine, entonces elegir otro proceso y ejecutarlo hasta que termine, y si sucesivamente.
2. Compartir el tiempo de procesador dando rodajas de tiempo a los procesos.

La primera opción es extremadamente injusta, pero evita muchos cambios de contexto. La segunda es más justa y adecuada para sistemas multiusuario.

Ejemplo:

```
program SalidaCaracteres;
process type imprimir (ch:char);
var i:integer;
begin
        for i:=1 to 5 do
          write (ch)
end;
var print1,print2,print3:imprimir;
begin
        cobegin
                print1('A');
                print1('B');
                print1('C');
        coend
end.
```

Si usamos la planificación justa, posibles salidas para el programa serían:

```
CCCACCAABBBAABB
CAABBCABCCABABC
AABBCCAABCCACBB
```

Pero si usamos una planificación injusta primero se imprimirán 5 A, luego 5 B y finalmente 5C.

Ejecución de un programa en Pascal-FC

Pascal-FC consiste en 2 programas, un compilador (pfccomp) y un intérprete. Se utiliza de la siguiente forma:

pfccomp <FicheroFuente> <ficheroListado> <FicheroObjeto> donde:

FicheroFuente: contiene el código en pascal-fc

ficheroListado: es el fichero donde se guarda un listado especial del programa junto a la información adicional como la tabla del símbolos o el código intermedio generado. Cuando se producen errores de compilación éstos pueden ser vistos en este fichero.

FicheroObjeto: es el fichero donde se guarda el resultado de la compilación y servirá de entrada al intérprete.

Con pfccomp ejemplo.pfc l o permite compilar el fichero ejemplo.pfc

Para el intérprete existen 2 modalidades: una justa y otra injusta. El intérprete que implementa la planificación justa se llama **pint** y el que implementa la planificación injusta **ufpint**. Los dos se utilizan de la misma forma:

pint <FicheroObjeto> <ficheroProblemas>

ufpint <FicheroObjeto> <ficheroProblemas>

donde, ficheroProblemas: es un fichero que solo se genera si existen problema en la ejecución del programa tales como un interbloqueo.

Para realizar las fases de compilación e interpretación en un paso, utilizando el programa **pfc** de la siguiente manera:

pfc <FicheroFuente> para una intérprete justo

pfc [-uf] <FicheroFuente> para una intérprete injusto

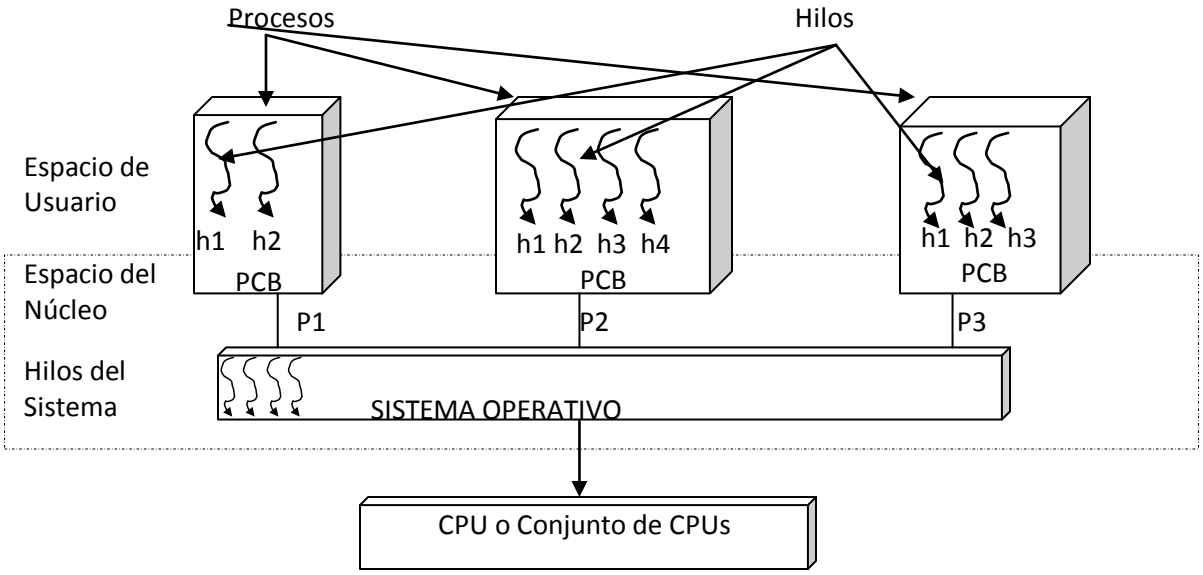
Hilos:

¿Qué es un hilo? De la misma manera que un sistema operativo puede ejecutar varios procesos al mismo tiempo ya sea por concurrencia o paralelismo, dentro de un proceso puede haber varios hilos de ejecución.

Un hilo puede definirse como cada secuencia de control dentro de un proceso que ejecuta sus instrucciones de forma independiente.

(Figura: sobre el hw subyacente (una o varias cpu's) se sitúa el sistema operativo, sobre éste se sitúan los procesos (P_i) que pueden ejecutarse concurrentemente y dentro de estos se ejecutan los hilos (h_i) que también pueden ejecutarse concurrentemente dentro de los procesos. Es decir tenemos concurrencia en dos niveles, una entre procesos y otra entre hilos de un mismo proceso). Si por ejemplo tenemos 2 procesadores, se podrían estar ejecutando al mismo tiempo el hilo 1 del proceso 1 y el hilo 2 del proceso 3, o podría ser el hilo 1 y 2 del proceso 1.

Los procesos son **entidades pesadas**, la estructura del proceso está en la parte del núcleo y cada vez que el proceso quiere acceder a ella, tiene que hacer algún tipo de llamada al sistema (consumiendo tiempo extra al procesador). Los cambios de contextos entre procesos son costosos en cuanto a tiempo de computación. Por el contrario la estructura de hilos reside en el espacio de usuario, con lo que un hilo es una **entidad ligera**. Y comparten información del proceso (datos, código, etc.) si un hilo modifica una variable del proceso, el resto de los hilo ven ese cambio cuando accedan a esa variable. El cambio de contexto entre hilos consumen poco tiempo de procesador, he ahí su éxito.



Concurrencia a dos niveles: procesos e hilos

Podemos hablar de dos niveles de hilos: 1- Aquellos que usaremos para programar y pueden crearse desde lenguajes de programación como Java, y 2- los hilos propios del sistema operativo, que sirven para dar soporte a los hilos de usuario.

Cuando se programa con hilos se hace uso de un API (Application Program Interface) proporcionado por el lenguaje, el SO o un tercero mediante una librería externa, el implementador de ese API será el que haya usado los hilos del sistema para dar soporte de ejecución a los hilos que se crean en los programas

Estándares de hilos:

Cada SO implementa los hilos del sistema como quiere, pero existen 3 librerías nativas de hilos que compiten por ser la mas usada: WIN32, OS/2 y POSIX. Las dos primeras solo corren bajo sus respectivas plataformas. Mientras que POSIX conocida como pthreads está pensada para todas las plataformas y está disponible para la mayoría de las implementaciones UNIX y Linux, también para VMS y AS/400.

Por su parte los hilos en Java están implementados en la máquina virtual Java (MVJ), que está construida sobre la librería de hilos nativa de la correspondiente plataforma. Java ofrece su API para manejar hilos.

Implementación de Hilos

Hay fundamentalmente dos formas de implementar hilos.

1. Es escribir una librería al nivel de usuario. Todas las estructuras y el código de las librerías estarán en espacio de usuario. La mayoría de las llamadas que se hagan desde la librería se ejecutarán en el espacio de usuario y no hará uso de las rutinas del sistema Implementa-

ción a nivel de Núcleo, la mayoría de las llamadas de la librería requerirán llamadas al sistema.

Ambos métodos pueden ser usados para implementar exactamente el mismo API.

Planificación de hilos:

Los sistemas operativos como Solaris definen el concepto de procesador lógico de tal forma que donde ejecutan los hilos de usuario es en n procesador lógico. En la terminología de Solaris a los procesadores lógicos se le denomina LWP (light-Weight Processes).

De esta forma vamos a tener una planificación a dos niveles. Un primer nivel para asignar los hilos de usuario a los procesadores lógicos y otro para asignar los procesadores lógicos al procesador o procesadores físicos. Los hilos de los usuarios compiten por los procesadores lógicos, (primer nivel de planificación), mientras que a su vez compiten por los hilos del sistema que son los que directamente se ejecutarán en los procesadores físicos (Segundo nivel de planificación).

Hay principalmente tres técnicas distintas para hacer la planificación de hilos sobre los recursos del núcleo (indirectamente sobre las distintas CPU´s):

Muchos hilos en un procesador lógico:

Conocido como el modelo **muchos a uno.** Todos los hilos creados en el espacio de usuario hacen turnos para ir ejecutándose en el único procesador lógico asignado a ese proceso. De esta forma un proceso no toma ventaja de una máquina con varios procesadores físicos. Otra desventaja es que cuando se hace una llamada bloqueante, por ejemplo de E/S, todo el proceso se bloquea. Es algo parecido a lo que ocurría con los procesos en Pascal-FC pero al nivel de hilo. Como ventaja podemos decir que en este modelo la creación de hilos y la planificación se hacen en el espacio de usuario al 100%, sin usar los recursos del núcleo. Por tanto, es más rápido.

Un hilo por procesador lógico:

Llamado, modelo uno a uno. Aquí se asignan un procesador lógico para cada hilo de usuario. Este modelo permite que muchos hilos se ejecuten simultáneamente en diferentes procesadores físicos, tiene el inconveniente de que la creación de hilos conlleva la creación de un procesador lógico. Como cada procesador lógico toma recursos adicionales del núcleo, uno está limitado en el número de hilos que puede crear. Utilizan este modelo Win21, OS/2, algunas implementaciones de POSIX y cualquier MVJ basadas en estas librerías.

Muchos hilos en muchos procesadores lógicos

Llamado muchos a muchos (estricto), el número de hilos es multiplexado en un número de procesadores lógicos igual o menor. Muchos hilos pueden correr en paralelo en diferentes CPUs, y las llamadas al sistema de tipo bloqueante no bloquean al proceso entero.

Modelos de dos niveles

O muchos a muchos (no estricto). Es como el anterior pero ofrece la posibilidad de hacer un enlace de un hilo específico con un procesador lógico. Probablemente sea el mejor modelo. Varios sistemas operativos (Digital Unix, Solaris, IRIX, HP-UX, AIX). En el caso de Java, las MVJ sobre estos sistemas operativos tienen la opción de usar cualquier combinación de hilos enlazados o no. La elección del modelo de hilos es una decisión al nivel de implementación de los escritores de la MVJ.

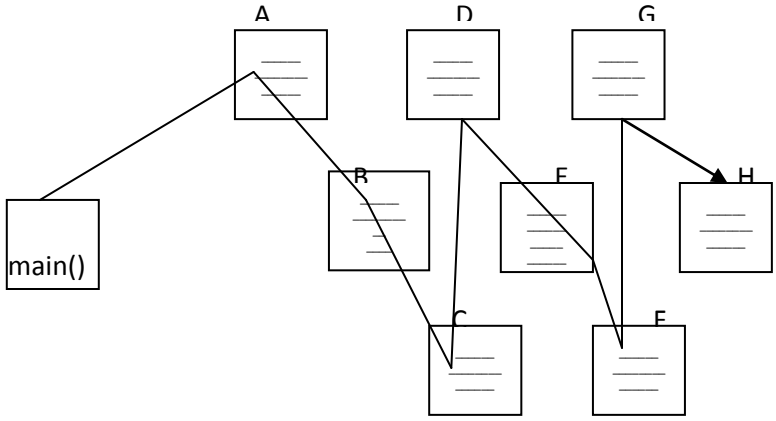
HILOS en JAVA

Java proporciona un API para el uso de hilos. Este API es bastante simple comparado con otras librerías de hilos como Posix.

Los hilos se representan mediante la clase *Thread*, que se encuentra en el paquete java.lang.thread. los métodos de esta clase junto con algunos métodos de la clase *Object* son los que permiten un manejo prácticamente completo de hilos en Java-

Hilos y objetos

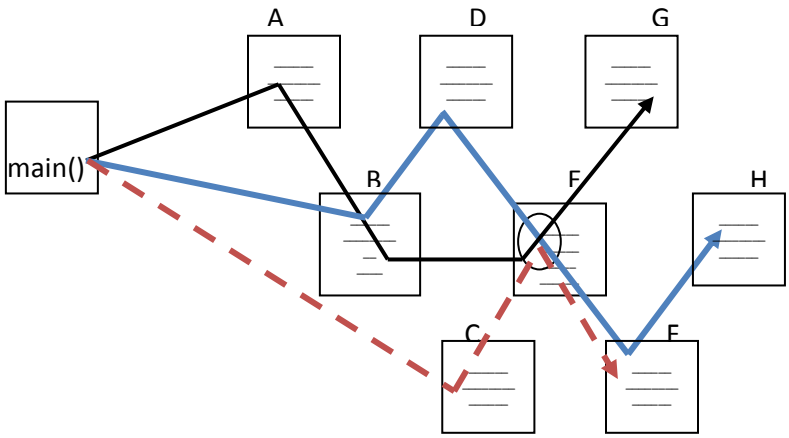
Al empezar un programa Java, hay un hilo de ejecución que es el indicado por el método *main()* denominado principal, si no se crean más hilos desde el hilo principal, tendremos algo como la siguiente figura, donde no solo existe un hilo de ejecución que va recorriendo los distintos objetos participantes en el programa según se vayan produciendo las distintas llamadas entre métodos de los mismos.



Varios objetos y un solo hilo

Sin embargo, si desde el hilo principal se crean por ejemplo dos hilos, podremos observar como el programa principal se ejecuta al mismo tiempo por tres sitios distintos (el hilo principal mas dos hilos creados). Los hilos pueden estar ejecutando código de diferentes objetos, código diferente en el mismo objeto o el mismo código en el mismo objeto al mismo tiempo.

La diferencia que existe entre un objeto y un hilo es que, un objeto es algo estático, tiene una serie de atributos y métodos, quién realmente ejecuta esos métodos es el hilo de ejecución. En ausencia de hilos solo hay un hilo que va recorriendo los objetos según se van produciendo las llamadas entre los métodos de los objetos. Puede darse el caso del objeto E de la siguiente figura, donde tres hilos de ejecución distintos estén ejecutando el mismo método al mismo tiempo.



Tres hilos “atravesando” varios objetos

Creación de Hilos

Existen dos formas de crear hilos:

1. Heredando de la clase Thread
2. Implementando la interfaz Runnable.

Heredando de Thread y redefiniendo el método run:

La clase Thread tiene un método especial cuya signatura es **public void run().** Cuando creamos una clase cuyas instancias queremos que sean un hilo, tendremos que heredar de clase Thread y redefinir este método. Dentro de este método se escribe el código que queremos que se ejecute cuando se lance a ejecutar el hilo. Se podría decir que main() es a un programa lo que run es a un hilo.

En el siguiente ejemplo, creamos una clase denominada ThreadConHerencia. Si queremos que los objetos pertenecientes a esta clase sean hilos, debemos extender de la clase Thread el constructor de la clase ThreadConHerencia recibe como parámetro una palabra a imprimir, en el método run() especificamos el código a ejecutar al lanzar el hilo: la impresión de la palabra 10 veces.

```
Class ThreadConHerencia extends Thread {
        String palabra;
public ThreadConHerencia (String _palabra) {
        palabra=_palabra;
}
public void run ( ) {
        for (int i:=0; i<10;i++)
                System.out.print (palabra);
}
```

Desde el programa principal creamos los hilos a y b:

```
public static void main(String [] args){
Thread a =new ThreadConHerencia (“hiloUno”);
Thread b =new ThreadConHerencia (“hiloDos”);
```

Para poner los hilos en ejecución se debe invocar el método start(), este método pertenece a la clase Thread. Se encarga de hacer algunas inicializaciones propias del hilo y posteriormente invoca al método run(). Es decir, invocando **start**, se ejecuta en orden los métodos start (heredado) y run (redefinido).
Deberemos añadir al programa principal las líneas invocando start.

```
a.start();
b.start();
System.out.println ("Fin del hilo principal")
```

Corriendo el programa, podemos observar cómo se intercalan las salidas de los tres hilos y entre ellos se pueden intercalar de cualquier forma.

Implementando la interfaz Runnable

En el siguiente código, implementamos la interfaz runnable para crear hilos del ejemplo anterior.
Creamos una clase denominada ThreadConRunnable, esta interfaz tiene un solo método con la signatura ***public void run ()***. Este método es el que, como mínimo, tenemos que implementar en esta clase.

```
Class ThreadConRunnable implements Runnable {
        String palabra;
public ThreadConRunnable (String palabra) {
        palabra=_palabra;
}
public void run ( ) {
        for (int i:=0; i<10;i++)
                System.out.print (palabra);
}
```

Hasta aquí se creó una clase, los objetos pertenecientes a esa clase no serán hilos, no se heredo de la clase Thread. Si queremos que un objeto de la clase ThreadConRunnable se ejecute en un hilo independiente, tendremos que crear un objeto de la clase Thread y pasarle como parámetro el objeto donde queremos que empiece su ejecución ese hilo. A continuación se crean dos objetos de la clase ThreadConRunnable (a,b) y dos hilos (t1 yt2) a los que se le pasa como parámetro los objetos a y b. posteriormente hay que invocar el método start() de la clase Thread que invoca al método run() de los objetos a y b respectivamente:

```
public static void main(String args[]){
ThreadConRunnable a =new ThreadConRunable ("hiloUno");
ThreadConRunnable b =new ThreadConRunable ("hiloDos");
Thread t1 =new Thread (a);
Thread t2 =new Thread (b);
t1.start();
t2.start();
System.out.println ("Fin del hilo principal")
```

"La segunda forma de crear hilos puede parecer más confusa. Sin embargo es más apropiada, debido a que como Java no hay herencia múltiple, la primera opción hipoteca la clase para que no pueda heredar de otras clases. Sin embargo con la segunda opción podemos hacer que el método

run() de una clase se ejecute en un hilo y además esta clase herede el comportamiento de otra clase".

Objeto autónomo en un hilo.

Sin importar que método se use para crear hilos, para que el objeto que implementa el método run se ejecute en un hilo depende del método cliente (main en nuestro caso). Es decir el programa principal tiene el control del control y creación de los hilos (tiene que lanzar a ejecución el nuevo hilo). Aun en la implementación de la interfaz runnable debe crearlo explícitamente. Sin embargo a veces es preferible un objeto autónomo que automáticamente se ejecute en un nuevo hilo, sin intervención del cliente (como en JAVA).

En el siguiente ejemplo vamos a construir un objeto éste se ejecuta automáticamente en un nuevo hilo. En el constructor creamos el hilo y lo lanzamos.

```
class Autothread implements Runnable {
    prívate Thread me_;

 public AutoThread() {
    me_=new Thread (this);
   me_ .start();
}
 public void run() {
   if (me_==Thread.currentThread())
     for (int i=0;i<10;i++)
       System.out.println ("Dentro de Autothread.run()");
}
 public static void main (string [] args) {
     AutoThread miThread = new AutoThread();
    For (int i= 0;i<10;i++)
       System.out.println ("Dentro de main()");
}
```

Como vemos en la implementación del método run() hay que controlar cuál es el hilo que se está ejecutando. Para ello nos servimos del método currentThread() de la clase Thread. Este método no devuelve una referencia al hilo que está ejecutando ese código. Esto se hace para evitar que cualquier método de un hilo distinto haga una llamada a run() directamente, como ocurre en el siguiente caso:

```
public static void main (string [] args) {
     AutoThread miThread = new AutoThread();
    miThread.run();
while (true)
       System.out.println ("Dentro de main()");
}
```

Estado de un hilo en Java

La siguiente figura muestra el ciclo de vida de un hilo y los métodos que provocan el paso de un estado a otro.

1. Nuevo es el estado de un hilo cuando es creado con operador new, al lanzarlo con start() pasa a listo (puede acaparar el procesador si el planificador lo permite), cuando obtiene el procesador pasa a ejecución, y puede pararse por diversos motivos:
2. Por hacer una operación de E/S pasara a bloqueado hasta completarse la operación
3. Por ejecutar sleep (milisegundos). Saldrá de este estado al cumplirse el tiempo especificado como parámetro.
4. Por intentar adquirir un cerrojo de un objeto.
5. Por ejecutar el método wait(). Saldrá de este estado cuando se ejecute el método notify() o notifyAll() por parte de otro hilo.

Puede observarse que el ciclo de vida de un hilo es parecido al de un proceso. El planificador de hilos de nuestras aplicaciones en JAVA se encuentra implementado en JMVJ.

Planificación y prioridades

Los modelos de hilos se caracterizan por:

1. Todos los hilos en JAVA tienen una prioridad y se supone que le planificador dará preferencia al hilo con prioridad mayor. Sin embargo no existe garantía que en un momento dado el hilo de mayor jerarquía se esté ejecutando.
2. Las rodajas de tiempo pueden ser aplicadas o no. Dependerá de la gestión de hilos que haga la librería sobre la que se implementa la maquina virtual java.

Se debe asumir que los hilos pueden intercalarse en cualquier punto en cualquier momento.

Nuestros programas no deben estar basados en la suposición de que vaya a haber un intercalado entre hilos. Si esto es un requisito en nuestra aplicación se deberá introducir el código necesario.

Prioridades:

Las prioridades de cada hilo en java están en un rango de 1 (MIN_PRIORITY) a 10 (MAX_PRIORITY). La prioridad de un hilo es inicialmente la misma que la del hilo que lo creó (prioridad 5 (NORM_PRIORITY). Los de menor prioridad van a ejejcutarse cuando estén bloqueados los de prioridad superior.

Las prioridades se pueden cambiar con método setPriority (nueva prioridad), la prioridad de un hilo en ejecución se puede cambiar en cualquier momento. El método getPriority() devuelve la prioridad.

Yield() hace que el hilo en ejecución ceda el paso de ejecución a otros que estén listos, el hilo elegido puede ser incluso el q cedió paso si su prioridad es mayor.

A continuación puede verse como se crean dos hilos (t1 y t2) y al primero de ellos se le cambia la prioridad. Esto hace que t2 acapare el procesador hasta terminar pues nunca será interrumpido por un hilo de menor prioridad.

```
public class A implements Runnable {
    String palabra;
   public A (String  _palabra) {
      palabra =_palabra;

 public void run() {
       for (int i=0;i<10;i++)
       System.out.println (palabra);
}
```

```
 public static void main (string [] args) {
    A a1= new A("a1");
    A a2=new A("a2");
   Thread t1 = new Thread(a1);
   Thread t2 = new Thread(a2);
    t1.start();
    t1.setPriority(1);
    System.out.println ("Prioridad de t1:"+t1.getPriority());
    t2.start();}
    System.out.println ("Prioridad de t2:"+t2.getPriority());
}
```

HILOS DAEMON

Antes de lanzar un hilo puede ser definido como Daemon, indicando que va hacer una ejecución continua para la aplicación como tarea de fondo. La máquina virtual abandonará la ejecución cuando todos los hilos que sean Daemon hayan finalizado su ejecución. Los hilos Deamon tienen la prioridad mas baja. Se usa el métodos setDaemon(true) para marcar un hilo como hilo demonio y se usa getDaemon() para probar ese indicador. por efecto, la cualidad de demonio se hereda desde el hilo que crea un nuevo hilo. No puede cambiarse después de haber iniciado un hilo.

Tipos de sincronización y su solución:

Los dos tipos de sincronización necesaria entre procesos concurrentes son: exclusión mutua y condición de sincronización.

Exclusión mutua:

Cuando los procesos deben utilizar un recurso no compartible, la sincronización necesaria entre ellos se denomina **Exclusión mutua**. Un proceso que accede a un recurso no compartible se encuentra en su sección crítica (es parte del código que utiliza un proceso cuando accede a un recurso no compartible y se ejecuta en exclusión mutua). Cuando un proceso está en su sección crítica el resto de los procesos no deben estar en sus secciones críticas, y deben esperar hasta que la sección crítica se libere.

Garantizar la exclusión mutua en la ejecución de las secciones críticas consiste en diseñar un ´protocolo de entrada y salida que sincronice la entrada de los procesos a sus secciones críticas. Un esquema sería:

Process P1	Process P2	Process Pi
….	….	….
Protocolo de entrada	Protocolo de entrada	Protocolo de entrada
Sección crítica	Sección crítica	Sección crítica
Protocolo de salida	Protocolo de salida	Protocolo de salida
….	….	….

Los protocolos de entrada y salida son porciones de código que permiten cumplir con las siguientes condiciones para que resuelvan el problema de exclusión mutua:

- **Exclusión mutua:** no pueden acceder dos procesos a la vez a la sección crítica. (Exclusión mutua hace referencia a la ejecución mutuamente exclusiva de las secciones críticas)
- **Limitación en la espera:** de cada proceso para acceder a la sección crítica; es decir, que ningún proceso espere indefinidamente para entrar en su sección crítica.
- **Progreso en la ejecución:** es decir que cuando un proceso quiera ejecutar su sección crítica pueda hacerlo si ésta está libre.

Condición de sincronización:

La podemos definir como la propiedad requerida de que un proceso no realice un evento hasta que otro proceso haya realizado una acción determinada.

Soluciones a los dos tipos de sincronización:

Los mecanismos que disponemos para implementar los distintos tipos de sincronización son:

1. Inhibición de las interrupciones.
2. Espera ocupada (busy waiting).
3. Semáforos.
4. Regiones críticas.
5. Regiones críticas condicionales
6. Monitores.
7. Operaciones de paso de mensajes send/ receive.
8. Llamadas a procedimientos remotos.
9. Invocaciones remotas

Estos mecanismos (exceptuando Inhibición de las interrupciones) están englobados en:

1. **Soluciones basadas en variables compartidas** (Espera ocupada, Semáforos, Regiones críticas, Regiones críticas condicionales, Monitores)
2. **Soluciones basadas en el paso de mensajes** (Operaciones de paso de mensajes send/ receive, Llamadas a procedimientos remotos, Invocaciones remotas)

   En el primer grupo se encuentran los mecanismos de espera ocupada, semáforos, regiones críticas, regiones críticas condicionadas y monitores. (estos mecanismo aparecen ordenados de menor a mayor nivel, es decir los 1° son mas complejos de usar. Las soluciones que se pueden alcanzar son distintas dependiendo del nivel por el cual aproximemos el problema.) la característica común en estos mecanismos es que independientemente del nivel del nivel, es que existe un problema elemental de exclusión mutua en el acceso concurrente a la misma posición de memoria. Este problema está resuelto a nivel hardware que serializa los accesos concurrentes.

   En el segundo grupo, se encuentran incluídas las operaciones de paso de mensajes send/receives, las llamadas a procesamientos remotos y las invocacines remotas. Las acciones atómicas de los procesos no son las operaciones de acceso a memoria, sino las sentencias de paso de mensajes.

   Vamos a abordar la primera solución propuesta para implementar la sincronización denominado Inhibición de interrupciones.