Programación de Sistemas y Concurrencia

Tema 4: Soporte a la Concurrencia en Lenguajes y Sistemas operativos

Grado en Ingeniería Informática

Grado en Ingeniería del Software

Grado en Ingeniería de Computadores

Concepto de proceso

- P || Q || R significa que la ejecución de P, Q y R puede solaparse en el tiempo.
- Cada proceso P, Q, R
  - representa una hebra de ejecución secuencial,
  - tiene asociado un procesador lógico que le permite evolucionar en su código
- El orden de la composición P||Q no es relevante:
  - || es conmutativo

Características de los procesos

- Estructura estática vs dinámica
- Jerarquía entre procesos
- Inicialización
- Terminación
- Representación de los procesos en los lenguajes de programación

Estructura estática vs dinámica

- Los lenguajes de programación pueden soportar una estructura de procesos
  - Estática: el número de procesos en ejecución está determinado en compilación
  - Dinámica: los procesos pueden crearse dinámicamente, por lo que el número total de procesos sólo puede conocerse en ejecución
  - Aunque la estructura sea dinámica, el número máximo de procesos está limitado por la de memoria de la máquina.

Jerarquías entre procesos

Posibilidad de anidar unos procesos dentro de otros.

- Existe una relación padre/hijo entre el proceso desde el que se crea a otro proceso, y el nuevo proceso creado.
- La consecuencia más importante de esta relación es que un proceso padre no puede terminar su ejecución hasta que lo hayan hecho todos sus hijos
  - El padre puede contener recursos que pueden estar siendo utilizados por sus hijos.
- Siempre existe un nivel de anidamiento mínimo: el programa principal es padre de todos los procesos del sistema creados.

Inicialización

- Para darle valores iniciales a un proceso pueden utilizarse dos métodos:
  - Paso de parámetros: como en los procedimientos o métodos
  - Paso de mensajes: el proceso padre, cuando crea un subproceso le envía un mensaje para inicializar sus variables
- Como distintos valores iniciales pueden dar lugar a distintos comportamientos, es natural introducir el concepto de *tipo de proceso*.
  - Podemos definir un patrón de comportamiento común y luego instanciarlo varias veces para obtener distintos procesos que aunque comparten el código, pueden comportarse de forma distinta dependiendo de sus valores iniciales.

Terminación

- Un proceso puede acabar su ejecución por distintas causas:
  - cuando se acaba su código (terminación con éxito)
  - cuando tiene un error durante su ejecución (terminación con fallo)
  - por la ejecución de una instrucción de autodestrucción (suicidio)
  - cuando otro proceso abortar su ejecución
- Un proceso puede no terminar
  - siendo éste su comportamiento esperado
  - como consecuencia de un error en su diseño

Creación y representación

- La creación de un proceso (el momento en que empieza a ejecutarse) puede ser
  - Explícita: se llama al proceso, como si fuera un procedimiento
  - Implícita: el proceso empieza a ejecutarse en cuanto lo hace el módulo o estructura en la que se encuentra anidado.
- La representación de los procesos puede ser
  - Explícita: el lenguaje proporciona un constructor específico para definir a los procesos
  - Implícita: Cualquier constructor del lenguaje puede ser considerado como un proceso (procedimiento, función, instrucción simple).

Creación y representación

- Creación explícita, representación implícita:
  - Concurrent Pascal
    - Si s_1,…s_N son instrucciones cualesquiera del lenguaje

      cobegin s_1;…;s_N coend;

      representa su ejecución concurrente
- Creación implícita, representación explícita
  - Ada

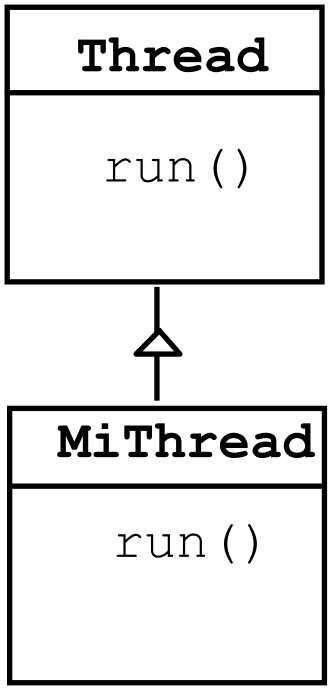
```
procedure Proc
  process P;
  begin …end;

  process Q;
  begin …end;
begin
  {Proc, P y Q están ejecutándose}
  …
end;
```

El modelo de proceso en Java

- Estructura dinámica de procesos.
  - Cualquier proceso puede crear uno nuevo
  - No hay límite en el número de procesos, salvo el impuesto por la memoria de la máquina
- Jerarquía arbitraria
  - Puede existir cualquier nivel de anidamiento entre procesos
- Inicialización
  - Mediante paso de parámetros en el constructor
- Terminación
  - Soporta todos los tipos de terminación, aunque el uso de la instrucción abort está desaconsejado
- Creación y representación explícitas
  - Existe un tipo proceso/hebra
  - Para que los procesos/hebras se ejecuten hay que llamarlos explícitamente.

El modelo de proceso/hebra en Java

- La clase **Thread** proporciona el soporte para manejar las hebras (que hemos llamado procesos hasta ahora).
- Las hebras pueden crearse y destruirse de forma dinámica (durante la ejecución del programa)

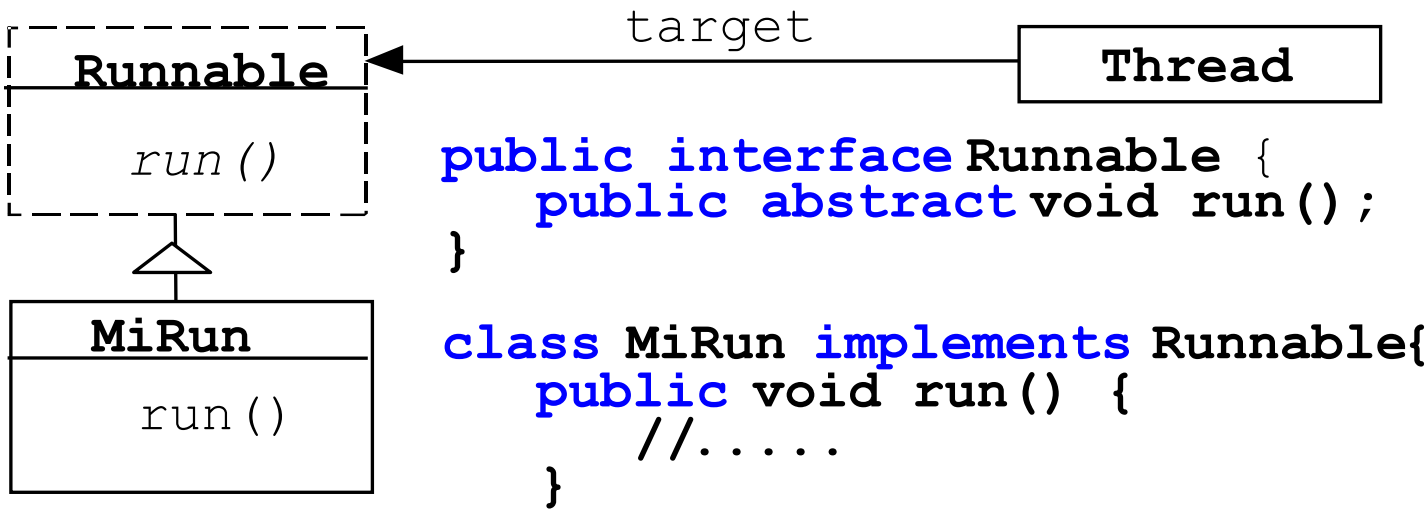


La clase Thread class ejecuta las instrucciones de su método run(). El código que realmente se ejecuta depende de la implementación de este método en la clase derivada.

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

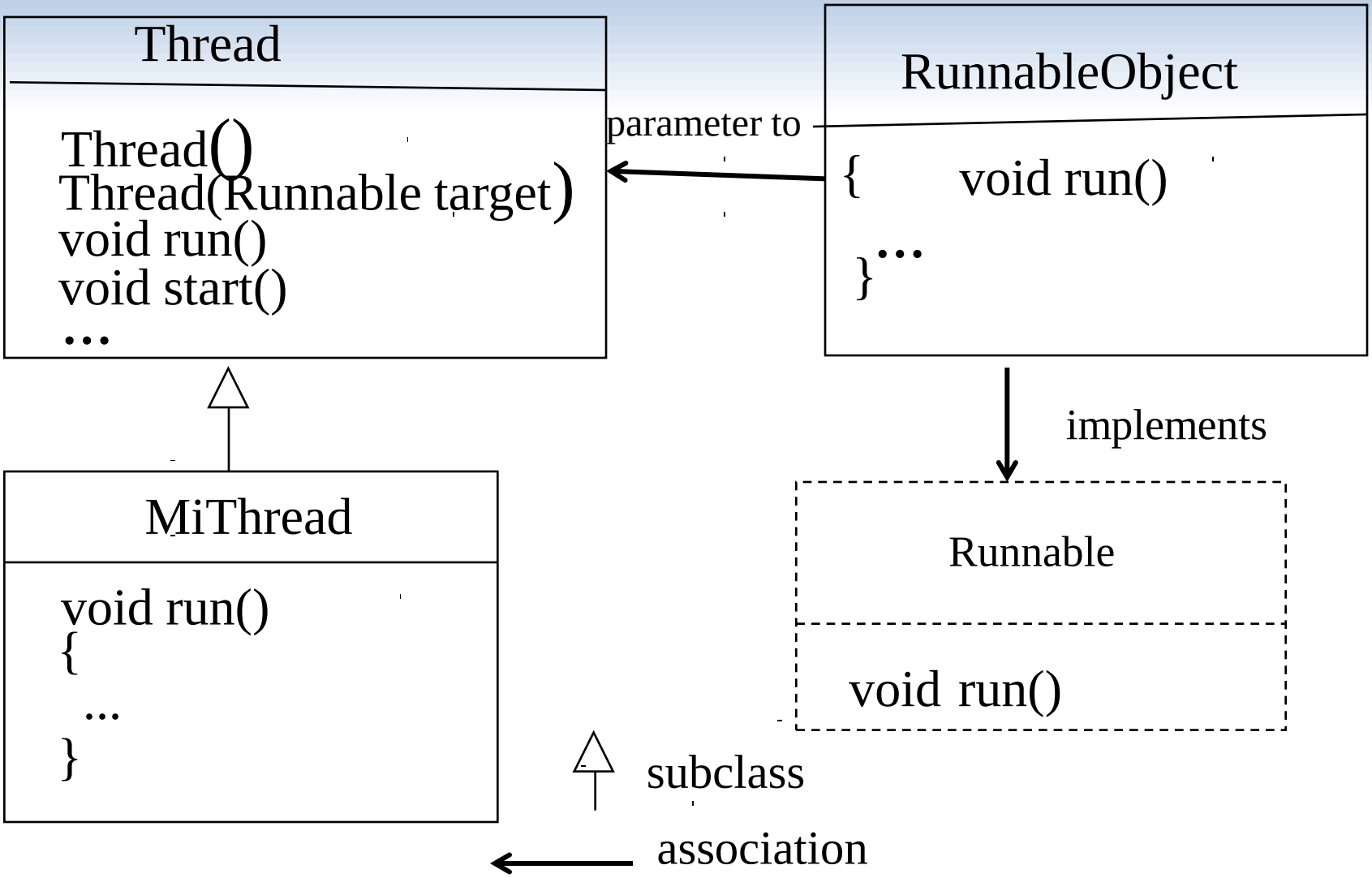
El modelo de proceso/hebra en Java

- Como Java no permite la herencia múltiple, si queremos que un objeto hebra herede de otra superclase, y de la clase Thread, el lenguaje porporciona la interfaz Runnable, que obliga a implementar el método run() y que permite la una clase que no deriva de Thread sea una hebra.
- Uno de los contructores de Thread permite pasarle como parámetro un objeto Runnable




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

class MiRun implements Runnable{
    public void run() {
        //.....
    }
}
```

El modelo de proceso/hebra en Java



La clase Thread

```
public class Thread extends Object
                implements Runnable {
  public Thread();
  public Thread(String name);
  public Thread(Runnable target);
  public Thread(Runnable target,
                String name);
  public Thread(Runnable target,
          String name, long stackSize);

  public void run();
  public void start();
  ...
}
```

La clase Thread

- Para crear una (hebra) Thread
  - Extendemos la clase Thread y sobreescribimos el método run()
  - O, creamos un objeto que implementa la interfaz Runnable, y le pasamos su referencia a un objeto de la clase Thread a través del constructor.

Un ejemplo simple

```
public class Esc implements Runnable{
        private char miLetra;
        public Esc(char l){
          miLetra = l;
        }
        public void run() {
          for (int i = 0; i<100; i++){
            System.out.print(miLetra);
          }
        }
}
```

```
public static void main(String[] args){
        Thread a = new Thread(new Esc('A'));
        Thread b = new Thread(new Esc('B'));
        Thread c = new Thread(new Esc('C'));
        a.start();b.start();c.start();
```

Todavía no hay ninguna hebra en ejecución

Un ejemplo simple

```
public class Esc implements Runnable{
        private char miLetra;
        public Esc(char l){
          miLetra = l;
        }
        public void run() {
          for (int i = 0; i<100; i++){
            System.out.print(miLetra);
          }
        }
}
```

```
public static void main(String[] args){
        Thread a = new Thread(new Esc('A'));
        Thread b = new Thread(new Esc('B'));
        Thread c = new Thread(new Esc('C'));
        a.start();b.start();c.start();
```

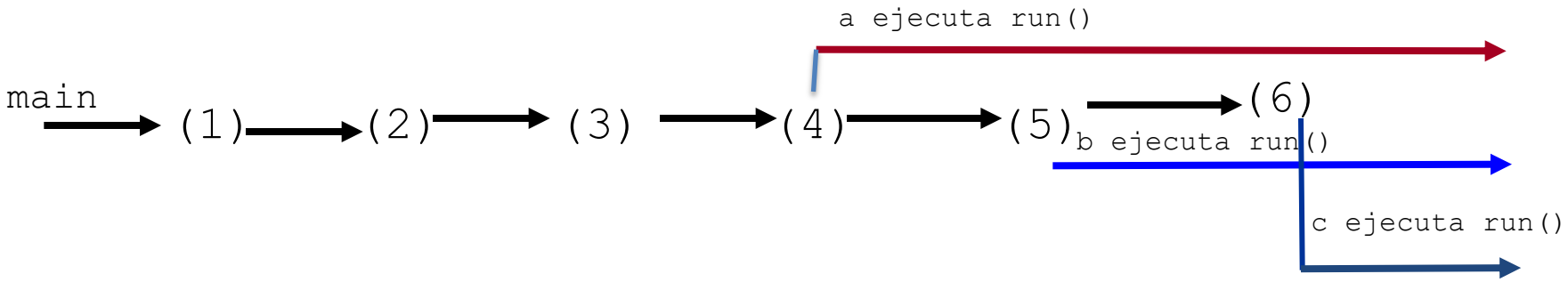
La llamada a start() hace que se ejecute el método run()

Las hebras a, b y c comienzan su ejecución cuando se llama a start()

Un ejemplo simple

```
public class Esc implements Runnable{
        private char miLetra;
        public Esc(char l){
          miLetra = l;
        }
        public void run() {
          for (int i = 0; i<100; i++){
            System.out.print(miLetra);
          }
        }
}
```

```
public static void main(String[] args){
  (1) Thread a = new Thread(new Esc('A'));
  (2) Thread b = new Thread(new Esc('B'));
  (3) Thread c = new Thread(new Esc('C'));
  a.start();b.start();c.start();
    (4)       (5)       (6)
}
```

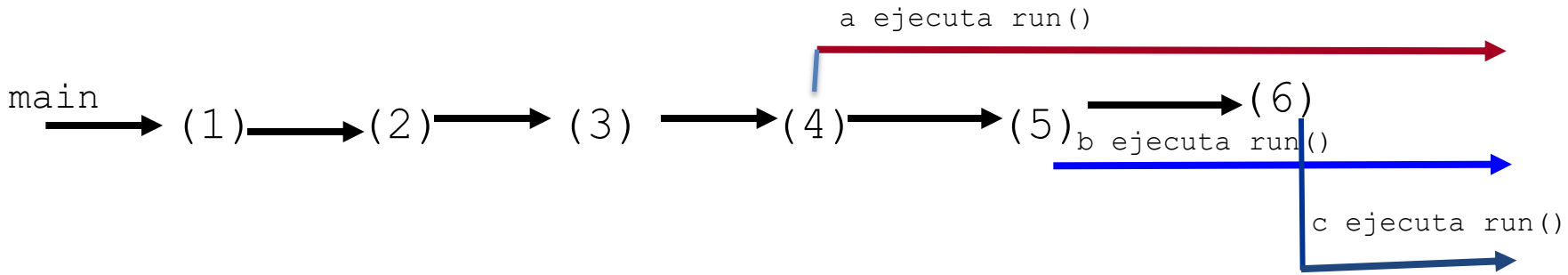


Cuando main llama a los tres procesos, las ejecuciones de main, a, b y c prosiguen de forma asíncrona

Un ejemplo simple

```
public class Esc implements Runnable{
        private char miLetra;
        public Esc(char l){
          miLetra = l;
        }
        public void run() {
          for (int i = 0; i<100; i++){
            System.out.print(miLetra);
          }
        }
}
```

```
public static void main(String[] args){
  (1) Thread a = new Thread(new esc('A'));
  (2) Thread b = new Thread(new esc('B'));
  (3) Thread c = new Thread(new esc('C'));
  a.start();b.start();c.start();
    (4)       (5)       (6)
}
```



La salida es una combinación de las letras A, B y C

BCCCCCCCCCCCAAAAAAAAAABBBBBBBBB

Otra alternativa usando Thread

```
public class Esc extends Thread{
      private char miLetra;
      public Esc(char l){
                miLetra = l;
      }
     public void run() {
         for (int i = 0; i<10; i++){
             System.out.print(miLetra);
          }
      }
}
```

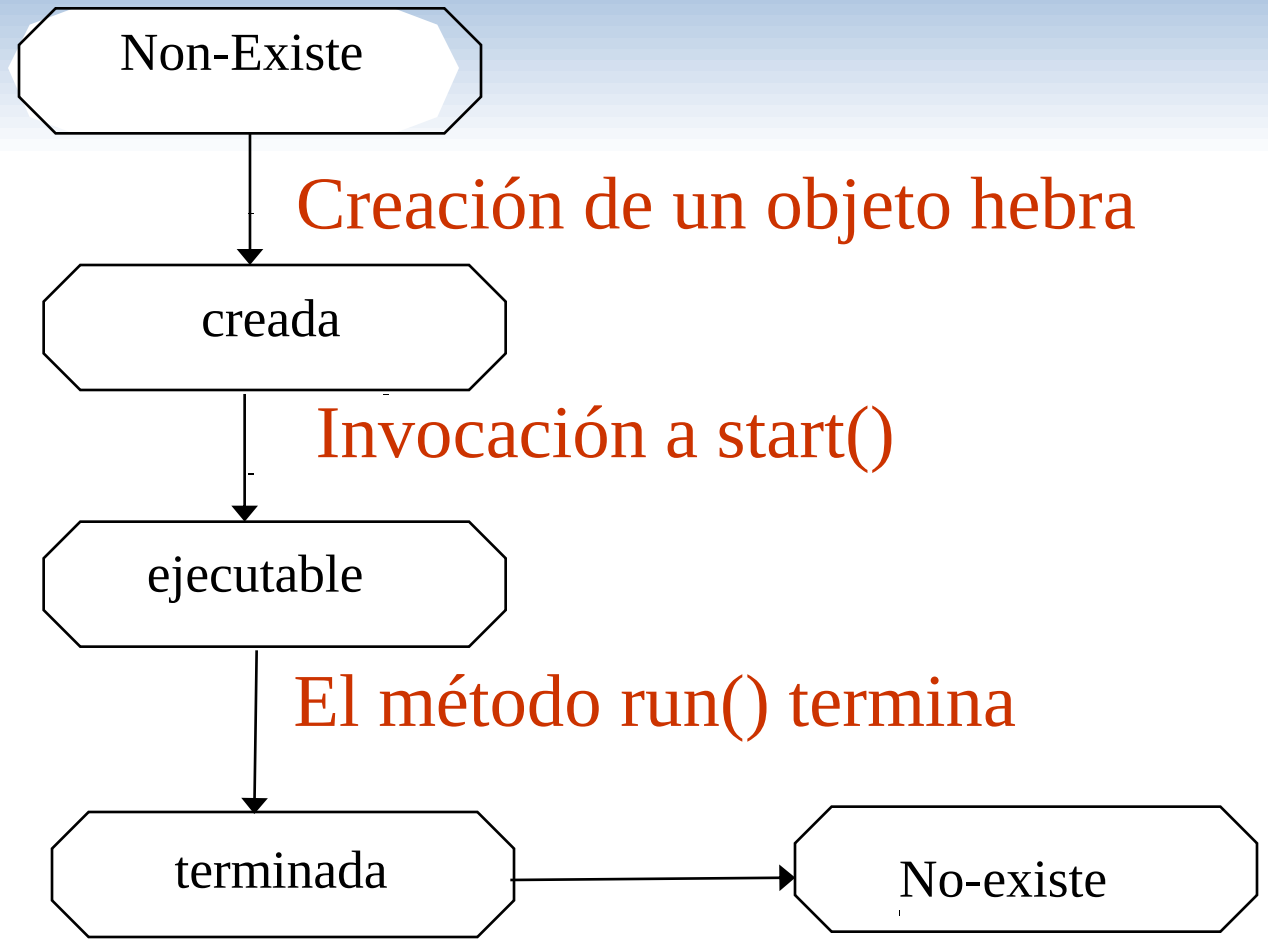
```
public static void main(String[] args){
                Esc a = new Esc('A');
                Esc b = new Esc('B');
                Esc c = new Esc('C');
                a.start();b.start();c.start();
 }
```

Advertencia

El método run() no debe llamarse directamente desde la aplicación. El sistema lo llama cuando se llama a start().

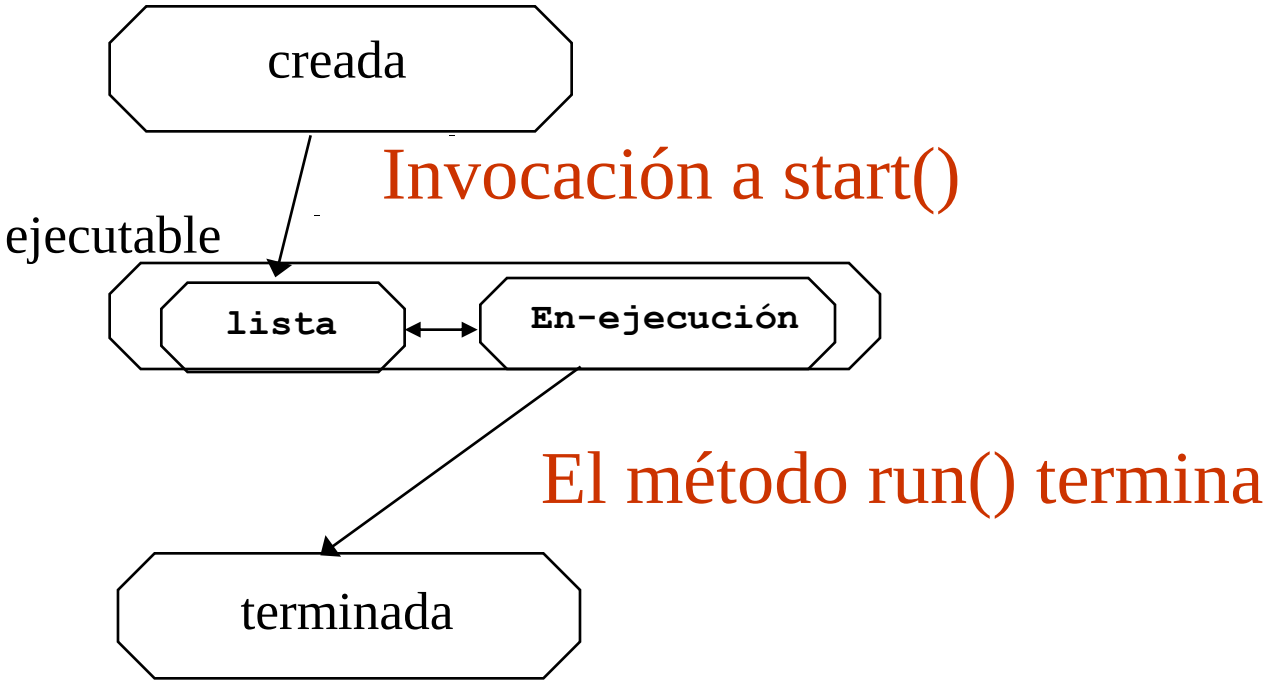
Si el método run() es llamado explícitamente, entonces el código se ejecuta de forma secuencial,
no concurrentemente.

Ciclo de vida de una hebra



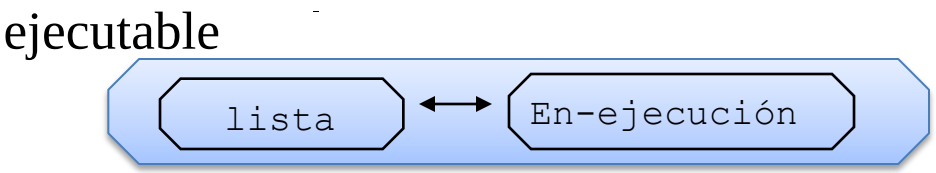
Estado ejecutable

- En un sistema monoprocesador, o en el que hay más hebras que procesadores sólo una, de entre las hebras ejecutables, puede estar ejecutándose, el resto debe esperar.



Estado ejecutable

- Una hebra pasa de lista a en-ejecución cuando el procesador se queda disponible
- Una hebra pasa del estado en-ejecución a lista
  - En un sistema de tiempo compartido (time-sharing) cuando se le ha acabado el tiempo de ejecución
  - Cuando se hace ejecutable una hebra con más prioridad.
  - El cambio de una hebra por otra en el procesador se denomina “cambio de contexto” y es un proceso con una carga computacional.
  - Hay que guardar los valores de todos los registros de la CPU, para que cuando la hebra vuelva al procesador, el cambio de contexto no haya interferido en su ejecución.
  - El cambio de contexto sólo puede realizarse entre la ejecución de dos instrucciones máquina, nunca durante la ejecución de una de ellas.

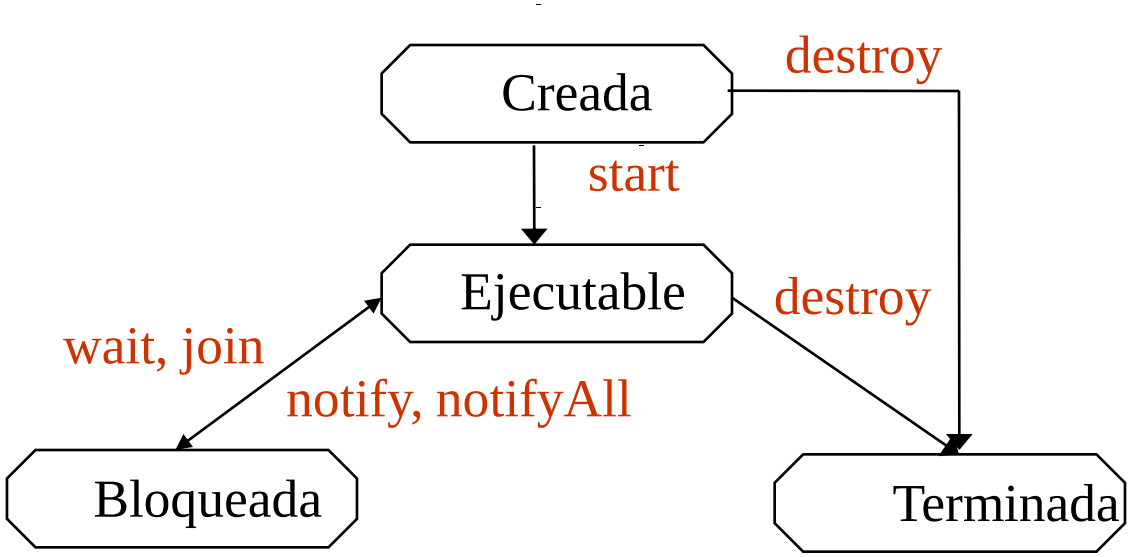


Clase Thread

```
public class Thread extends Object
                implements Runnable {
  ...
  public static void yield()

  public static void sleep(long millis)
          throws InterruptedException;
  public static void sleep(long millis, int nanos)
          throws InterruptedException;
}
```

Estado Bloqueado

- Una hebra se encuentra en un estado bloqueado cuando no puede continuar su ejecución hasta que ocurra “algún evento”.
  - Por ejemplo, que un recurso que necesita quede disponible
- Este estado es imprescindible para la sincronización de las hebras



Terminación de una hebra

- Cuando el método run termina su ejecución
  - Normalmente
  - Como resultado de una excepción no manejada
- Cuando la hebra llama al método stop el método run termina, pero antes
  - Se liberan los locks que tuviera cogidos
  - Se ejecutan las cláusulas finally que quedan
  - A continuación el objeto hebra puede ser limpiado por el recolector de basura de java
  - stop no es seguro, porque puede dejar al resto de las hebras en un estado inconsistente. En la actualidad se desaconseja su uso
- Cuando se llama al método destroy
  - En este caso, la hebra termina abruptamente, sin realizarse ninguna acción adicional. Como es un método muy inseguro algunas máquinas javas no lo implementan. Se desaconseja su uso.

Hebras Demonio

- En java las hebras pueden ser de dos tipos: hebras de usuario o hebras demonio (daemon)
- Las hebras demonio son hebras que proporcionan servicios al resto de las hebras del sistema y, por lo tanto, deben permanecer vivas mientras esté viva cualquier otra hebra
- Cuando todas las hebras usuario han terminado, las hebras demonio pueden acabar y el programa principal también termina
- El método setDaemon permite convertir en demonio una hebra. Debe llamarse antes de que la hebra comience a ejecutarse.

Hebras Demonio

```
public class Thread extends Object
                implements Runnable {
  ...
  public void destroy(); // DEPRECATED

  public final boolean isDaemon();
  public final void setDaemon(boolean on);

  public final void stop();  // DEPRECATED
}
```

El método join

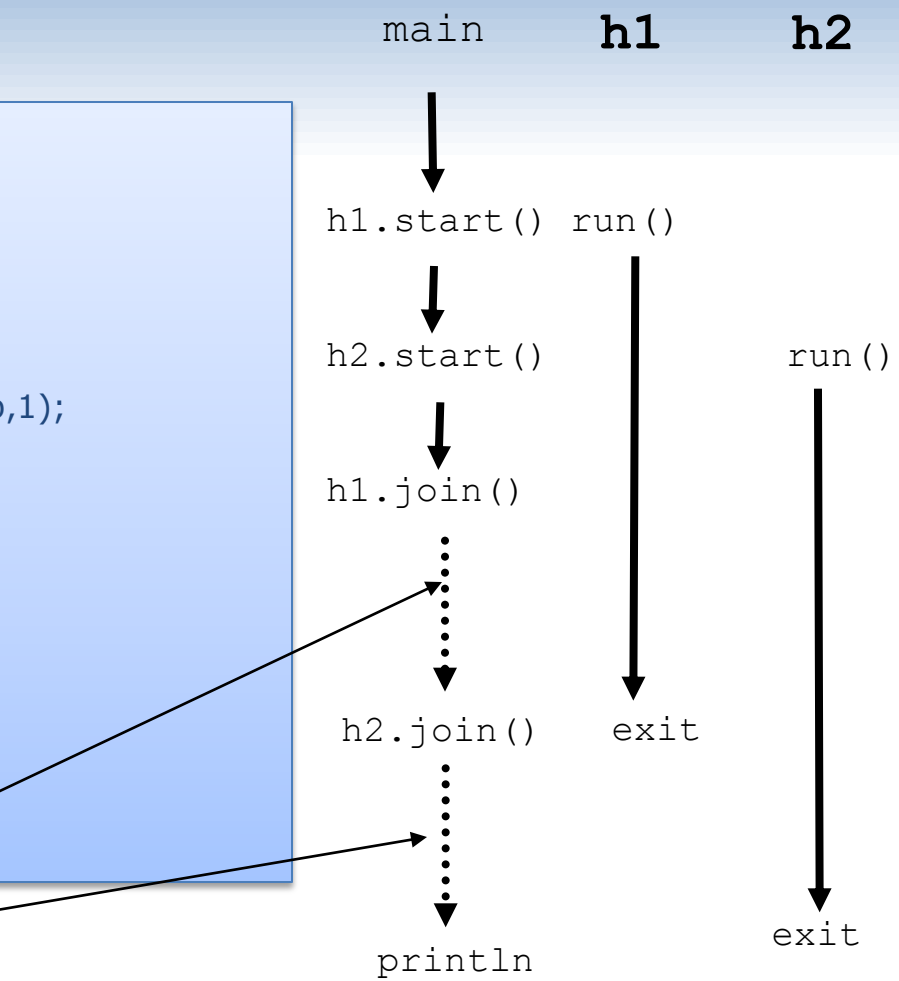
- Una hebra puede esperar (en estado bloqueado) a que otra hebra termine llamando a método join sobre esta otra hebra.

```
class Hebra extends Thread{
    private int[] v;
    private int inic,fin;
    private int id;
    private boolean[] b;
    public Hebra(iint[] v,nt inic,int fin, boolean[] b,int id){
        this.inic = inic; this.fin = fin;
        this.id= id;
        this.v= v;
        this.b = b;
     }

    public void run(){
        int i = inic;
        boolean escero = true;
        while (escero && i < fin){
            escero = v[i] == 0;
            i++;
         }
        b[id] = escero;
     }
}
```

```
public static void main(String[] args) {
    Random r = new Random();
    int[] v;
    boolean[] b = {true,true};
    v= new int[r.nextInt(10)];
    for (int i = 0; i<v.length;i++){
      v[i]= r.nextInt(2);
      System.out.print(v[i]+"  ");
    }
    Hebra h1 = new Hebra(v,0,v.length/2,b,0);
    Hebra h2 = new Hebra(v,v.length/2,v.length,b,1);
    h1.start();
    h2.start();
    try{
        h1.join();
        h2.join();
    } catch (InterruptedExeption ie){ }
    System.out.println(b[0] && b[1]);
}
```

El método join

```
public static void main(String[] args) {
    ……
    Hebra h1 = new Hebra(v,0,v.length/2,b,0);
    Hebra h2 = new Hebra(v,v.length/2,v.length,b,1);
    h1.start();
    h2.start();
    try{
        h1.join();
        h2.join();
    } catch (InterruptedException e){ }
    System.out.println(b[0] && b[1]);
  }
}
```



main bloqueado

El método isAlive

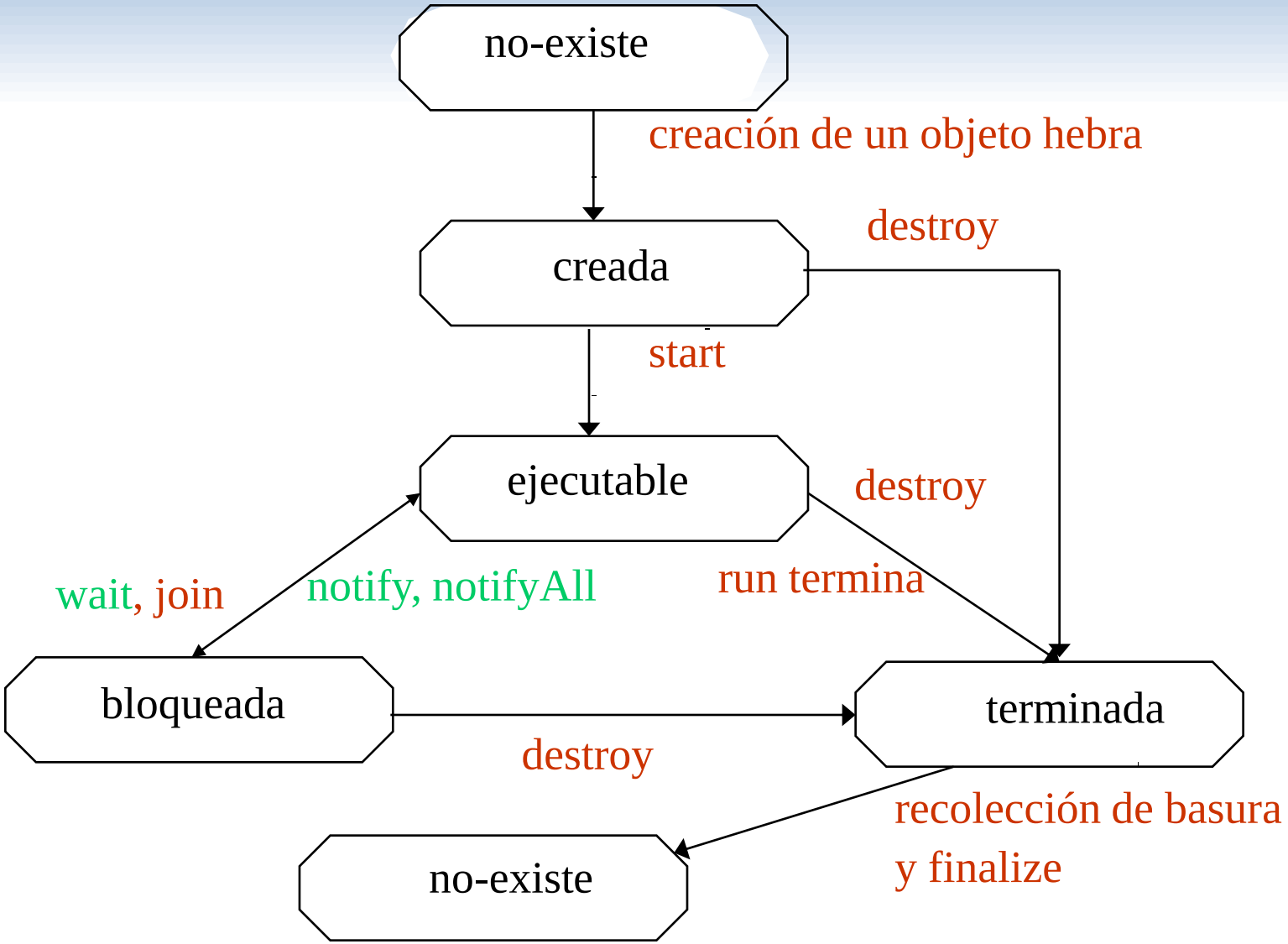
- Podemos utilizar el método isAlive para saber si una hebra todavía no ha terminado su ejecución
- Pero a diferencia de join, isAlive consume ciclos del procesador

```
public static void main(String[] args) {
        ……
      Hebra h1 = new Hebra(v,0,v.length/2,b,0);
      Hebra h2 = new Hebra(v,v.length/2,v.length,b,1);
      h1.start();
      h2.start();
      while (h1.isAlive()); //espera activa
      while (h2.isAlive()); //espera activa
      System.out.println(b[0] && b[1]);
   }
}
```

Clase Thread

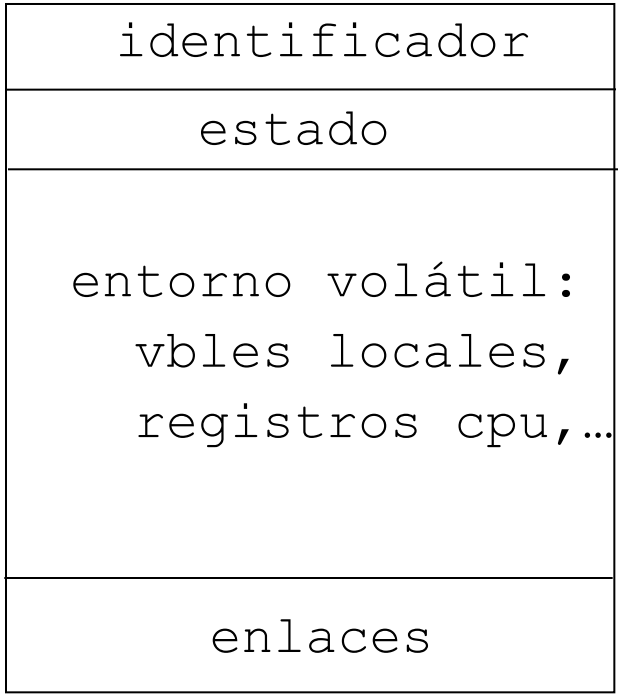
```
public class Thread extends Object
                implements Runnable {
  ...

  public final boolean isAlive();

  public final void join()
                  throws InterruptedException;

  public final void join(long millis)
                throws InterruptedException;

  public final void join(long millis, int nanos)
                throws InterruptedException;
}
```

Ciclo de vida completo



Ejecución de una programa concurrente

- Ejecutar un programa concurrente es más complejo que uno secuencial.
  1. Hay que representar internamente cada objeto hebra:



Ejecución de una programa concurrente

- Ejecutar un programa concurrente es más complejo que uno secuencial.
  1. hay que representar internamente cada objeto hebra
  2. hay que decidir en cada momento a qué hebra le toca ejecutarse (scheduler o planificador)
     - El criterio que utiliza el planificador se denomina política de planificación
- Los programas deben ejecutarse correctamente con independencia de la política de planificación subyacente.
  - En principio, no debería ser necesario conocer cómo la máquina ejecuta nuestro código.
  - Sin embargo, algunas propiedades de justicia (que veremos más adelante) pueden verse afectadas por esta política.

Ejecución de una programa concurrente

- La especificación del lenguaje java no define una forma particular de planificar las hebras. Sólo indica que las hebras deberían planificarse utilizando una política basada en las prioridades de las hebras (las hebras con mayor prioridad deberían ejecutarse antes).
- Sin embargo, algunas implementaciones de la máquina virtual podrían no seguir esta recomendación, o llevarla a cabo de forma diferente. Como consecuencia

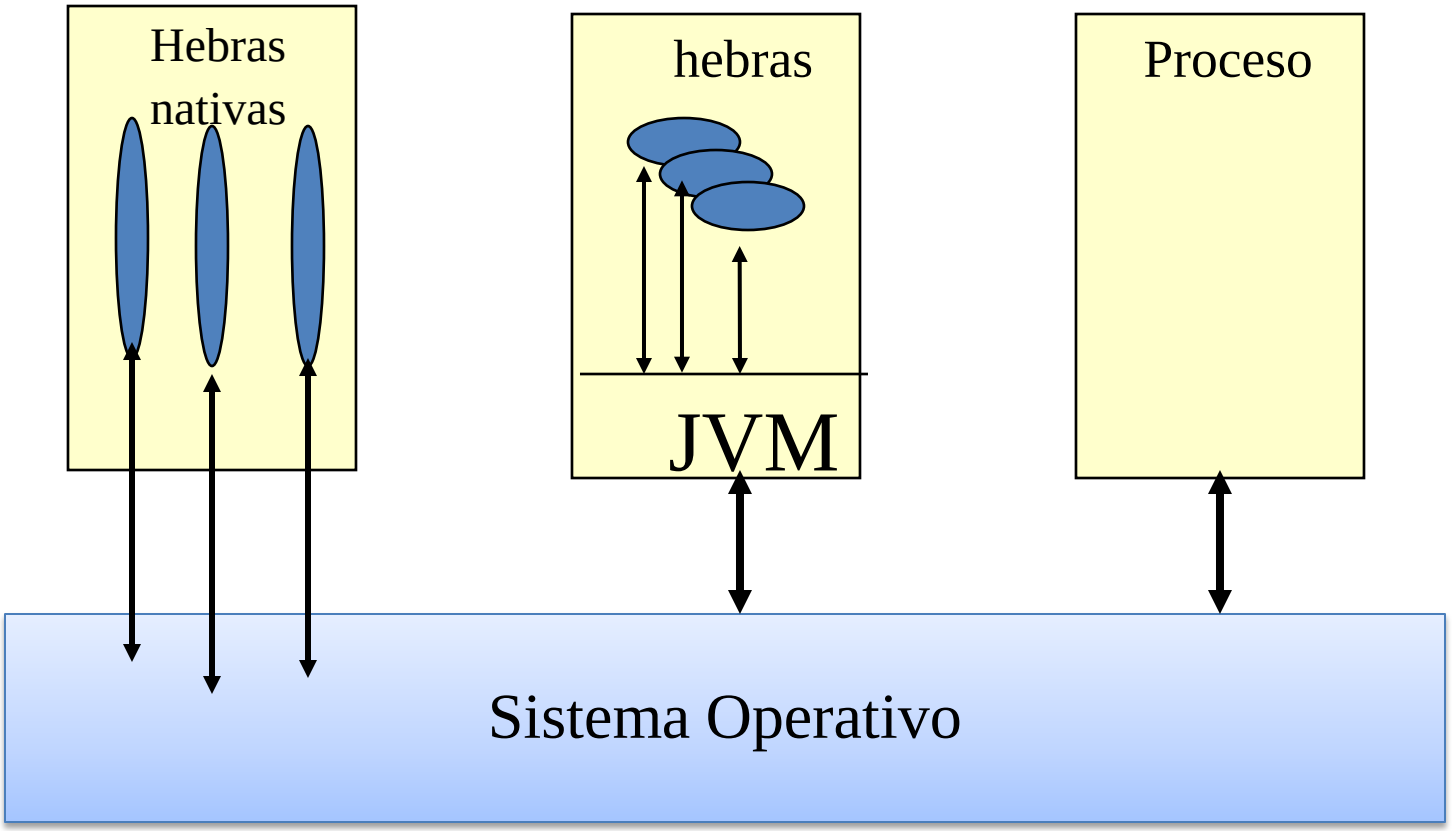
  el orden de ejecución de las hebras no está garantizado, depende de la plataforma de ejecución

- Además si todas las hebras tienen la misma prioridad, esta recomendación no implica nada.

Ejecución de un programa concurrente

- En java las hebras pueden ser de dos tipos
  - Green threads: son hebras que están implementadas sobre la máquina virtual, que emula la ejecución concurrente implementando su propio planificador. En la actualidad, muy pocas implementaciones de java utilizan este tipo de hebras.
  - Native threads: son hebras que el sistema operativo reconoce como tales. Hay una relación 1-1 entre las hebras java y las que el sistema operativo ejecuta. En este caso, la planificación de las hebras es realizada directamente por el sistema subyacente.

Procesos vs Hebras

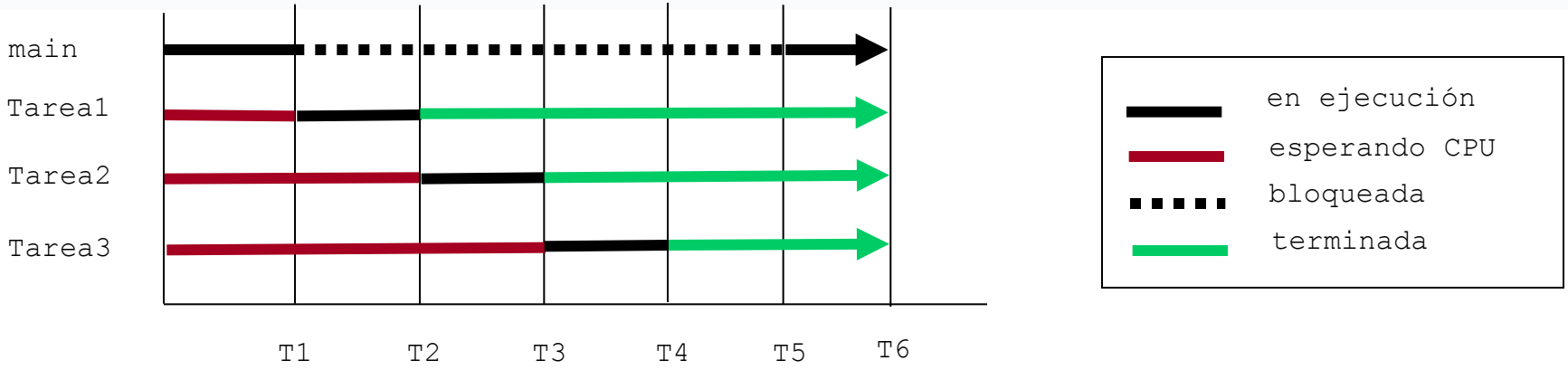


Tipos de Planificación: Ejemplo

```
public class Task extends Thread{

    private long n;
    private int id;
    public Task(long n,int id){
        this.n = n;
        this.id = id;
    }
    private long fib(long n){
        if (n == 0) return 0L;
        if (n == 1) return 1L;
        return fib(n-1)+fib(n-2);
    }

    public void run(){
       Date d = new Date();
       DateFormat df = new SimpleDateFormat("HH:mm:ss:SSS");
        long inicio = System.currentTimeMillis();
       d.setTime(inicio);

       System.out.println("Empieza la tarea "+id+" a las "+df.format(d));

       fib(n);

        long fin = System.currentTimeMillis();
       d.setTime(fin);

        System.out.println("La tarea "+id+" termina a las "+
                df.format(d)+". Ha tardado "+(fin-inicio)+" milisegundos");
    }
  ....
}
```

```
public static void main(String[] args){
      Task[] t = new Task[4];
      for (int i = 0;i<4;i++){
          t[i] = new Task(40,i);
          t[i].start();
       }
}
```

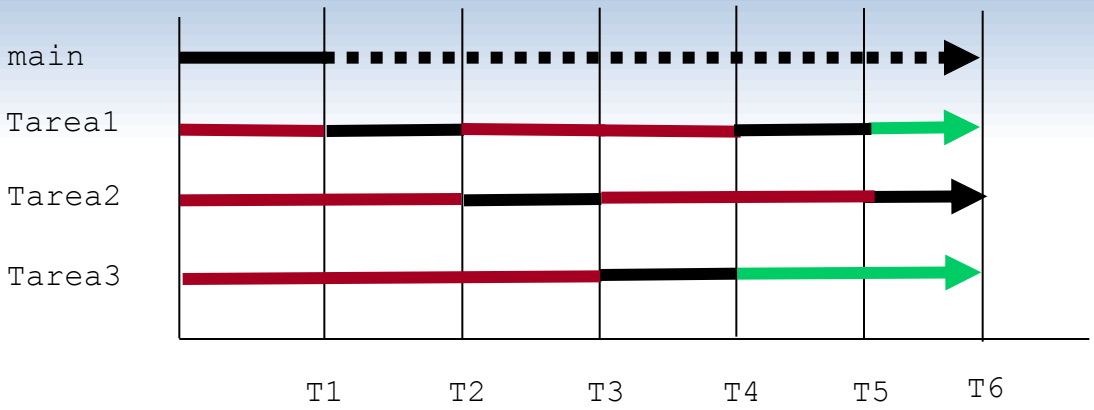
Tipos de Planificación: Ejemplo 1

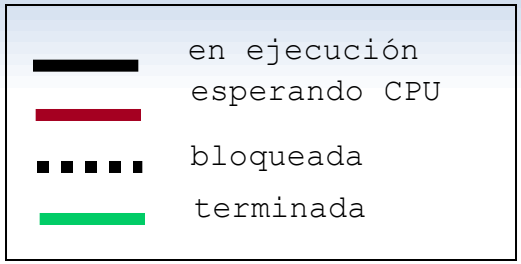



```
Empieza la tarea 0 a las 12:40:39:312
La tarea 0 termina a las 12:40:42:125. Ha tardado 2813 milisegundos
Empieza la tarea 1 a las 12:40:42:125
La tarea 1 termina a las 12:40:44:921. Ha tardado 2796 milisegundos
Empieza la tarea 2 a las 12:40:44:921
La tarea 2 termina a las 12:40:47:703. Ha tardado 2782 milisegundos
Empieza la tarea 3 a las 12:40:47:718
La tarea 3 termina a las 12:40:50:500. Ha tardado 2782 milisegundos
```

La ejecución parece secuencial porque el planificador ha dejado ejecutarse cada tarea todo el tiempo que ha necesitado

Tipos de Planificación: ejecución 2






```
Empieza la tarea 0 a las 12:51:13:031
Empieza la tarea 2 a las 12:51:13:031
Empieza la tarea 3 a las 12:51:13:046
Empieza la tarea 1 a las 12:51:13:046
La tarea 3 termina a las 12:51:18:687. Ha tardado 5641 milisegundos
La tarea 1 termina a las 12:51:18:750. Ha tardado 5704 milisegundos
La tarea 0 termina a las 12:51:18:765. Ha tardado 5734 milisegundos
La tarea 2 termina a las 12:51:18:781. Ha tardado 5750 milisegundos
```

La ejecución se intercala. El planificador asigna un tiempo de ejecución
A cada hebra y cuando se acaba cambia de hebra.
Esta planificación se llama time-slicing, y al cambio de hebra en la CPU
cambio de contexto (context switch)

Hebras vs Objetos

- En java una instancia de la clase **Thread** es un objeto, y como tal podemos pasar una referencia a ese objeto a métodos.
- Cualquier hebra que tiene una referencia a otra hebra puede ejecutar cualquier método del objeto Thread de la otra hebra.
- El **objeto** Thread **no** es la **hebra**.
- El objeto Thread equivale al conjunto de datos y métodos que encapsulan información sobre la hebra.
- Cualquier otra hebra puede acceder a esos datos y métodos.
- Por lo tanto, el código de la clase a la que pertenece un objeto no nos dice nada sobre las hebras que están ejecutando sus métodos, o examinando sus datos.
- Incluso si un objeto es de una clase que extiende a Thread, puede haber miles de hebras ejecutando su código.

Identidad de una hebra

- Podemos saber qué objeto hebra en particular está ejecutando un método utilizando el método currentThread de Object.
- Como este método es estático, hay sólo uno para todas las instancias, y podemos llamarlo utilizando la clase Thread.

```
public class Thread extends Object
                implements Runnable {
  ...
  public static Thread currentThread();
  ...
}
```

Ejemplo

```
class UnaHebra extends Thread{
    private int id;

    public UnaHebra(int id){
        this.id = id;
    }
    public  void quienSoy(){
            System.out.println(Thread.currentThread());
    }

    public String toString(){
        return "Hebra " +id;
    }

    public void run(){
        for (int i = 0; i<Iter ; i++){
            quienSoy();
        }
    }
}
```

```
class OtraHebra extends Thread{
        private int id;
        private UnaHebra h;
        public OtraHebra(int id, UnaHebra h){
            this.id = id;
            this.h = h;
        }


        public String toString(){
            return "OtraHebra " +id;
        }

        public void run(){
            for (int i = 0; i<Iter; i++){
                h.quienSoy();
            }
        }
}
```

```
public static void main(String[] args) {
        UnaHebra h= new UnaHebra(0);
        OtraHebra o = new OtraHebra(1,h);
        h.start();
        o.start();
}
```

objetos

hebras

Referencias

- Concurrency: State Models & Java Programs
  Jeff Magee, Jeff Kramer, Ed. Willey
- Concurrent Programming
  Alan Burns, Geoff Davies, Ed. Addison Wesley
- Concurrent and Real Time Programming in Java
  Andy Wellings, Ed. Willey