Sistemas Concurrentes y Distribuidos.
Seminario 2. Hebras en Java.

Dpt. Lenguajes y Sistemas Informáticos
ETSI Informática y de Telecomunicación
Universidad de Granada

Curso 16-17

Índice

Sistemas Concurrentes y Distribuidos.
Seminario 2. Hebras en Java.

---

1 Hebras en Java
2 Creación de hebras en Java
3 Estados de una hebra Java
4 Prioridades y planificación.
5 Interacción entre hebras Java. Objetos compartidos
6 Compilando y ejecutando programas Java

Introducción

Este seminario es una breve introducción a la programación con hebras en el lenguaje Java.

- El objetivo es conocer las principales formas de crear y gestionar y hebras en Java, con objeto de desarrollar los ejemplos de la práctica 2.
- Se mostrarán las distintas formas de crear hebras en Java, los distintos estados de una hebra Java y cómo controlar la prioridad de las hebras.
- Se verá un mecanismo para asegurar la exclusión mutua en el acceso concurrente a bloques de código Java.

Enlaces para acceder a información complementaria

- Aprenda Java como si Estuviera en primero.
- Tutorial de Java.
- Manual de Java.
- Información sobre la clase `Thread`.

Sección 1
Hebras en Java

Concepto de hebra en Java

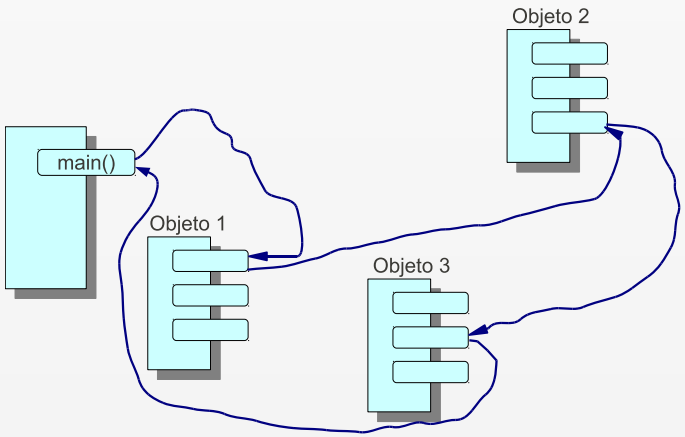
- La mayoría de las implementaciones de Java incluyen un compilador que genera código intermedio independiente de la máquina (llamado `bytecode`) más un intérprete de dicho código intermedio.
- El bytecode es interpretado por un proceso denominado **Máquina Virtual Java (JVM)**.
- Las hebras Java se ejecutan dentro de la JVM y comparten los recursos de este proceso del sistema operativo.
- Java ofrece una interfaz para manejar hebras.
- Una hebra Java es una instancia de una clase que implementa la interfaz `Runnable`, y/o instancias de una subclase de la clase `Thread`.
- Los métodos de las clases `Thread` y `Object` (definidas en el paquete `java.lang`) son los que hacen posible la gestión y sincronización de hebras en Java.

Ejecución de una hebra en Java

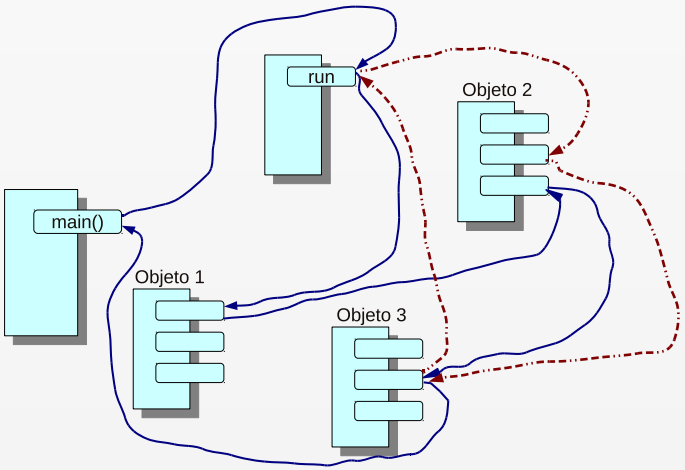
a) Por defecto, al ejecutar un programa Java, tenemos una hebra asociada al método `main()`, que va recorriendo distintos objetos conforme se invocan los correspondientes métodos.

b) Dentro de la hebra `main`, se pueden crear varias hebras que pueden ejecutar métodos del mismo o de diferentes objetos en el mismo o en diferente momento.

a)



b)



Sección 2
Creación de hebras en Java

Creación de hebras en Java

Para ejecutar una hebra en Java es necesario definir (al menos) una clase *C* que contenga un método **`run`**. Cada hebra ejecuta el código de dicho método para una instancia de *C*. Esto se puede implementar de dos formas:

- *C* es una clase derivada de la clase **`Thread`**:
  - **`Thread`** es una clase predefinida en el lenguaje.
  - Se impide que *C* sea derivada de otra clase distinta de **`Thread`**.
- *C* es una clase que implementa la interfaz **`Runnable`**:
  - **`Runnable`** es un interfaz predefinido en el lenguaje.
  - *C* puede extender (ser derivada de) una clase base cualquiera.
  - Se requiere adicionalmente la creación de una instancia de **`Thread`** por cada hebra a ejecutar. Esta instancia guarda una referencia a la instancia de *C*.

Nosotros usaremos la segunda opción precisamente por su mayor flexibilidad, a pesar de que es ligeramente más complejo. Habrá tantas clases *C* como sea necesario.

Uso de `Runnable` y `Thread`

Por cada hebra a ejecutar es necesario crear, además de una instancia r de C, una instancia t de **`Thread`** (t contiene una referencia a r). Hay dos posibilidades:

- El objeto r y después t se crean de forma independiente en la aplicación como dos objetos distintos.
- La clase C incluye como variable de instancia el objeto t (el objeto r tiene una referencia a t, o bien decimos que el objeto **`Runnable`** *encapsula* al objeto **`Thread`**).

nosotros usaremos la segunda opción ya que fuera del código de C solo tendremos que gestionar un objeto C en lugar de dos de ellos (uno C y otro **`Thread`**).

Creación y ejecución de una hebra.

Definiremos una clase **`TipoHebra`**, la cual

- puede ser derivada de una clase base **`Base`** cualquiera (no es necesario).
- tiene una variable de instancia pública de nombre, por ejemplo, **`thr`**, de la clase **`Thread`**.
- en su constructor crea el objeto **`thr`**.

Para ejecutar una de estas hebras es necesario:

1. Declarar una instancia **`obj`** de la clase **`TipoHebra`** (internamente se crea **`obj.thr`**).
2. Usar el método **`start`** de **`Thread`** sobre **`obj.thr`**, es decir, ejecutar **`obj.thr.start()`**.

La nueva hebra comienza la ejecución concurrente de **`obj.run()`**.

Ejemplo de una hebra

La definición de **TipoHebra** puede, a modo de ejemplo, ser así:

```
1  class TipoHebra implements Runnable // opcionalmente: extends ....
2  {
3    long siesta  ;       // tiempo que duerme la hebra
4    public Thread thr ; // objeto hebra encapsulado
5
6    public TipoHebra( String nombre, long siesta )
7    { this.siesta = siesta;
8      thr = new Thread( this, nombre );
9    }
10   public void run()
11   { try
12     { while ( true )
13       { System.out.println( "Hola, soy "+thr.getName() );
14         if ( siesta > 0 ) Thread.sleep( siesta );
15       }
16     }
17     catch ( InterruptedException e )
18     { System.out.println( "me fastidiaron la siesta!" );
19     }
20   }
21 }
```

Programa principal

Lanza una hebra y después hace **join** (la espera) y **getName**.

```
23 class Principal1
24 {
25   public static void main( String[] args ) throws InterruptedException
26   {
27     if ( args.length < 1 )
28     {  System.out.println( "Error: falta valor de 'siesta'" );
29        System.exit(1);
30     }
31     long siesta = Long.parseLong( args[0] ) ;
32     TipoHebra obj = new TipoHebra("hebra 'obj'", siesta); // crear hebra
33
34     obj.thr.start();       // lanzar hebra
35     Thread.sleep( 100 ); // la hebra principal duerme 1/10 sec.
36     obj.thr.join();        // esperar a que termine la hebra
37   }
38 }
```

Sección 3
Estados de una hebra Java

Estados de una hebra Java

- **Nueva**: Cuando el objeto hebra es creado, por ejemplo con:

```
Thread thr = new Thread (a);
```

- **Ejecutable**: Cuando se invoca al método **start** de una nueva hebra:

```
thr.start ();
```

- **En ejecución**: Cuando la hebra está ejecutándose en alguna CPU del computador. Si una hebra en este estado ejecuta el método **yield**, deja la CPU y pasa el estado ejecutable:

```
Thread.yield ();   // metodo static de la clase Thread
```

- **Bloqueada**: Una hebra pasa a este estado:
  - cuando llama al método **`sleep`**, volviendo al estado ejecutable cuando retorne la llamada:

```
Thread.sleep (tiempo_milisegundos}; // metodo estático
```

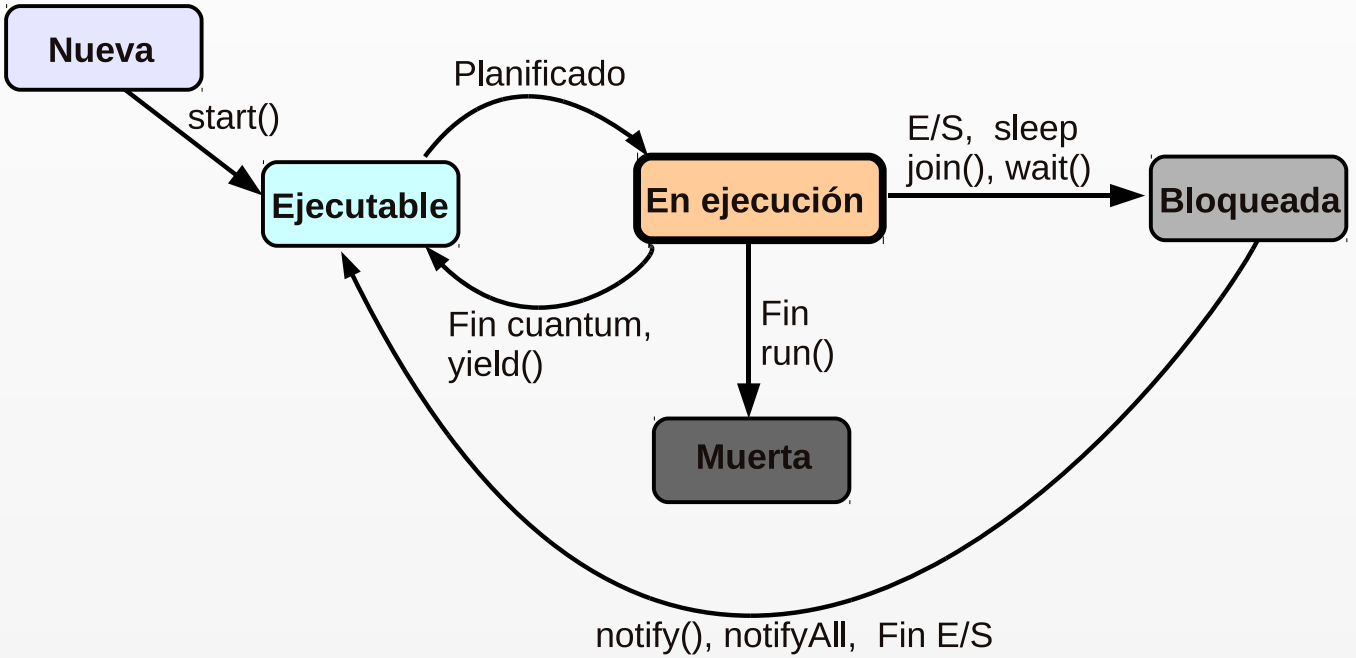
  - cuando llama a **`wait`** dentro de un bloque o un método sincronizado, volviendo al estado ejecutable cuando otra hebra llame a **`notify`** o a **`notifyAll`**
  - cuando llama a **`join`** para sincronizarse con la terminación de otra hebra que aún no ha terminado:

```
thr.join ();
```

  - cuando ejecuta alguna operación de e/s bloqueante.
- **Muerta**: Cuando su método **`run`** termina.

Transiciones entre estados

El siguiente diagrama muestra de forma resumida las diferentes transiciones entre estados y los métodos que los provocan:



Sección 4
Prioridades y planificación.

4.1. Un ejemplo más completo

Prioridad de una hebra

- Cada hebra tiene una prioridad que afecta a su planificación. Una hebra hereda la prioridad de la hebra que la creó. La prioridad es un valor entero entre **`Thread.MIN_PRIORITY`** y **`Thread.MAX_PRIORITY`** (dos constantes).
- La prioridad actual de una hebra puede obtenerse invocando el método **`getPriority`**, y se modifica con **`setPriority`**. En el ejemplo creamos una hebra y decrementamos su prioridad antes de llamar a **`start`**:

```
thr = new Thread( obj );
thr.setPriority( thr.getPriority()-1 );
thr.start();
```

Planificación de hebras

- El planificador de hebras de Java asegura que la hebra ejecutable con mayor prioridad (o una de ellas, si hay varias) pueda ejecutarse en la CPU. Si una hebra con mayor prioridad que la actual pasa a estar en estado ejecutable, la hebra actual pasa al estado ejecutable, y la hebra ejecutable de mayor prioridad es ejecutada (se apropia de la CPU).
- Si la hebra actual invoca a **`yield`**, se suspende, o se bloquea, el planificador elige para ejecución la próxima hebra de mayor prioridad.
- Otro aspecto de la planificación es el *tiempo compartido*, es decir la alternancia en el uso de la CPU por parte de las hebras ejecutables de mayor prioridad.

Subsección 4.1
Un ejemplo más completo

Ejemplo de un vector de hebras.

El este ejemplo se crea un array de hebras de acuerdo con el esquema visto en la primera sección. Cada hebra es una instancia de **Dormilona**:

```
44  class Dormilona implements Runnable
45  {
46    int vueltas = 0 ; // número de veces que duerme (0 == infinitas)
47    int siesta  = 0 ; // tiempo máximo que duerme cada vez
48    public Thread thr ; // objeto hebra encapsulado
49
50    public Dormilona( String p_nombre, int p_vueltas, int p_siesta )
51    {
52      siesta  = p_siesta ;
53      vueltas = p_vueltas ;
54      thr     = new Thread( this, p_nombre ) ;
55    }
56
57    //...
```

Ejemplo de un vector de hebras (2)

El método **run** contiene el código que ejecutan todas las hebras:

```
58    //...
59    public void run()
60    { try
61      { Random random = new Random(); // crea generador de números aleatorios
62        // dormir un numero de veces igual a "vueltas"
63        for ( int i=0 ; i < vueltas || vueltas == 0 ; i++ )
64        { // imprimir nombre
65          System.out.println( "Vuelta no."+i+", de " +thr.getName());
66          // duerme un tiempo aleatorio entre 0 y siesta-1 milisegundos
67          if ( siesta > 0 )
68            Thread.sleep( random.nextInt( siesta ) );
69        }
70      }
71      catch( InterruptedException e )
72      { System.out.println( Thread.currentThread().getName()+
73                                    ": me fastidiaron la siesta!");
74      }
75    } // fin run
76  } // fin de la clase Dormilona
```

Ejemplo de un vector de hebras (4)

El método main lee los parámetros.

```
79  class Principal2
80  {
81    public static void main( String[] args )
82    { try
83      { if ( args.length < 3 )
84        { System.out.println( "falta num_hebras, t_max_siesta, vueltas" );
85          System.exit( 1 );
86        }
87        int nHebras = Integer.parseInt( args[0] );
88        int siesta  = Integer.parseInt( args[1] );
89        int vueltas = Integer.parseInt( args[2] );
90
91        System.out.println( "nHebras = "+nHebras+", vueltas = "+vueltas+
92                            ", tsiesta = "+siesta );
93        //....
```

Ejemplo de un vector de hebras (5)

Finalmente, se crean las hebras en el vector **vhd** y se lanzan todas

```
94        //....
95
96        Dormilona[] vhd = new Dormilona[nHebras] ; // crea vector de hebras
97
98        for ( int i =0 ; i < nHebras ; i++ )
99        { String nombre = "Dormilona no."+i ;
100         vhd[i] = new Dormilona(nombre,vueltas,siesta); // crear hebra i.
101         vhd[i].thr.start();                            // comienza su ejec.
102       }
103       // la hebra principal duerme durante un segundo
104       Thread.sleep( 1000 );
105       System.out.println( "main(): he terminado de dormir" );
106
107       // esperar a que terminen todas las hebras creadas
108       for( int i = 0 ; i < nHebras ; i++ )
109         vhd[i].thr.join();
110     }
111     catch( InterruptedException e )
112     { System.out.println( "main(): me fastidiaron la siesta!" );
113     }
114   }
115 }
```

Sección 5
Interacción entre hebras Java. Objetos compartidos

5.1. Implementando exclusión mutua en Java. Código y métodos sincronizados.
5.2. Ejemplo: Cálculo de múltiplos

Interacción entre hebras Java. Objetos compartidos

- En muchos casos, las hebras de un programa concurrente en Java han de interactuar para comunicarse y cooperar.
- La forma más simple en la que múltiples hebras Java pueden interactuar es mediante un objeto cuyos métodos pueden ser invocados por un conjunto de hebras. Invocando los métodos de este *objeto compartido* las hebras modifican el estado del mismo.
- Dos hebras se pueden comunicar si una escribe el estado del objeto compartido y otra lo lee. Asimismo, la cooperación entre hebras se puede llevar a cabo mediante la actualización de alguna información encapsulada en un objeto compartido.
- La ejecución concurrente entre hebras Java supone un entrelazamiento arbitrario de instrucciones atómicas que puede dar lugar a estados incorrectos de los objetos compartidos por varias hebras.

Subsección 5.1
Implementando exclusión mutua en Java. Código y métodos sincronizados.

Exclusión mutua en Java. Código y métodos sincronizados.

- En Java, cda objeto tiene un cerrojo interno asociado. El cerrojo de un objeto solamente puede ser adquirido por una sola hebra en cada momento.
- El cualificador **`synchronized`** sirve para hacer que un bloque de código o un método sea protegido por el cerrojo del objeto.
- Para ejecutar un bloque o un método sincronizado, las hebras deben adquirir el cerrojo del objeto, debiendo esperar si el cerrojo ha sido ya adquirido por otra hebra.
- Si **`obj`** es una referencia a un objeto (de cualquier clase), la siguiente construcción hace que las hebras tengan que adquirir el cerrojo del objeto para ejecutar el bloque de código sincronizado.

```
synchronized( obj )
{
    // bloque de codigo sincronizado
}
```

Si todas las secciones críticas están en el código de un único objeto, podremos utilizar **`this`** para referenciar al objeto cuyo cerrojo vamos a utilizar:

```
synchronized( this )
{   // bloque de codigo sincronizado
}
```

Podemos hacer que el cuerpo entero de un método sea código sincronizado:

```
tipo metodo( ... )
{ synchronized( this )
  {   // codigo del metodo sincronizado
  }
}
```

La siguiente construcción es equivalente a la anterior:

```
synchronized tipo metodo( ... )
{   // codigo del metodo sincronizado
}
```

Subsección 5.2
Ejemplo: Cálculo de múltiplos

Clase contador de número de múltiplos

Programa multihebra para calcular el número de múltiplos de 2 y 3 entre 1 y 1000 ambos incluidos.

- Una hebra contará múltiplos de 2 y otra de 3.
- Para llevar la cuenta se crea un único objeto compartido de la clase **`Contador`**, que encapsula un valor entero.
- La clase **`Contador`** incluye un método para incrementarlo y otro para obtener el valor.
- El uso compartido requiere usar esas métodos en exclusión mutua.

Clase contador de número de múltiplos

La clase **Contador** puede declararse así:

```
1  class Contador
2  { private long valor;
3    public Contador( long inicial )
4    { valor = inicial ;
5    }
6    void retrasoOcupado() // ocupa la CPU durante cierto tiempo
7    { long tmp = 0 ;
8      for( int i = 0 ; i < 100000 ; i++ )
9        tmp = tmp*i-tmp+i ;
10   }
11   public synchronized void incrementa () // incrementa valor en 1
12   { long aux = valor ; // hace copia local del valor actual
13     retrasoOcupado() ; // permite entrelazado cuando no se hace en EM
14     valor = aux+1 ;     // escribe el valor compartido (incrementado)
15   }
16   public synchronized long getvalor () // devuelve el valor actual
17   {  return valor;
18   }
19 }
```

Hebras que cuentan los múltiplos

Cada una de las hebras incrementa el contador cuando encuentra un múltiplo:

```
21 class Hebra implements Runnable
22 { int numero ;                    // cuenta múltiplos de este número
23   public Thread thr ;             // objeto encapsulado
24   private Contador cont;          // contador de número de múltiplos
25
26   public Hebra( String nombre, int p_numero, Contador p_contador )
27   { numero = p_numero;
28     cont   = p_contador;
29     thr = new Thread( this, nombre );
30   }
31   public void run ()
32   { for ( int i = 1 ; i <= 1000 ; i++ )
33       if (i % numero == 0)
34         cont.incrementa();
35   }
36 }
```

¿ que pasaría si los dos métodos de **Contador** no fueran **synchronized** ?

Programa principal

Lanza dos hebras e imprime la cuenta calculada y la correcta:

```
38 class Multiplos
39 { public static void main( String[] args )
40   { try
41     { Contador contH = new Contador(0); // contador usado por la hebras
42       Hebra[] vc = new Hebra[2] ;
43       vc[0] = new Hebra( "Contador2 ", 2, contH);
44       vc[1] = new Hebra( "Contador3 ", 3, contH);
45       vc[0].thr.start(); vc[1].thr.start(); // lanza hebras
46       vc[0].thr.join();  vc[1].thr.join();  // espera terminación
47       System.out.println("Resultado hebras  : "+contH.getvalor());
48       long contV = 0 ;  // contador de verificacion
49       for ( int i = 1 ; i <= 1000 ; i++ )
50       { if ( i % 2 == 0 ) contV = contV + 1 ;
51         if ( i % 3 == 0 ) contV = contV + 1 ;
52       }
53       System.out.println("Resultado correcto: " + contV);
54     }
55     catch (InterruptedException e)
56     { System.out.println ("ha ocurrido una excepcion.");
57     }
58   }
59 }
```

Sección 6
Compilando y ejecutando programas Java

Compilar programas Java

Los programas fuente se encuentran en archivos de extensión `.java`. Cada uno de ellos contendrá una o mas definiciones de clases. Para compilar un fuente java (en un archivo `principal.java`, por ejemplo), hay que escribir:

```
javac principal.java
```

Si el programa `principal` hace mención a clases definidas en otros archivos, el compilador intentará compilarlos también, aunque se podría hacer manualmente antes:

```
javac otro1.java
javac otro2.java
javac ....
```

esto generará un archivo objeto (con *bytecode*) por cada clase definida en `principal.java` o en los otros archivos.

Ejecutar programas Java

Los archivos con *bytecode* tienen el nombre de la clase y la extensión `.class`. Una de las clases compiladas debe contener el método principal (`main`), sea esa clase *C*:

- Al compilar, el *bytecode* (código compilado) de dicha clase quedará, lógicamente, en un archivo de nombre *C*`.class`
- Para ejecutar el programa, se debe usar la orden `java`, que invoca al intérprete de *bytecode*.
- Se debe dar como argumento el nombre de la clase que contiene el método `main`, sin extensión, es decir, debemos escribir

```
java C
```

el intérprete buscará el archivo *C*.class (que debe contener la implementación de `main`), y lo ejecutará.

Java en las aulas de la ETSIIT

En las aulas de la ETSIIT hay distintas versiones de Java en distintas distribuciones de Linux. Para poder usar las órdenes `java` y `javac`, es necesario indicar donde se encuentran los correspondientes archivos ejecutables, para ello podemos extender la variable de entorno `PATH`:

- En **Ubuntu 12**

```
export PATH=/fenix/depar/lsi/java/jdk1.5.0linux/bin:.:${PATH}
```

- En **Ubuntu 14**

```
export PATH=/usr/local/jdk1.8.0_31/bin:.:${PATH}
```

(comprobado en Noviembre de 2016)