Threads

Antonio Espín Herranz

Contenidos

- Clase Thread
- Interface Runnable
- Sincronización
- Problema del Productor / Consumidor.

Thread

- Es la forma que tiene java de proporcionar la programación **multihilo**.
- Un programa multihilo contiene dos o mas partes que se pueden ejecutar de forma concurrente.
- Cada parte del programa se denomina Thread (hilo).
- Los hilos pueden compartir datos o recursos.

Tipos de multitarea

- Basada en procesos.
  - Un proceso es un programa ejecutándose. El proceso es la unidad mas pequeña. Permite al ordenador ejecutar dos o mas programas concurrentemente.
- Basada en hilos.
  - Parte de un programa que se puede ejecutar. Un hilo es la unidad más pequeña. Un solo programa puede realizar dos o mas tareas simultáneamente.

Ejemplos de tipos de multitarea

- Basada en procesos
  - En un ordenador estamos ejecutando un compilador de java y un editor de textos. Actúa sobre tareas generales. Sobre carga mas la CPU, requiere mas recursos. Es mas pesada.

- Basada en hilos:
  - Un editor de textos puede dar formato al texto a la vez que se está imprimiendo el texto. Cada acción la realizaría un hilo distinto. Actúa sobre los detalles.

Prioridades de los hilos

- Java asigna una prioridad a cada hilo para saber como se trata ese hilo con respecto a los demás.
- Son números enteros.
- La prioridad de un hilo se utiliza para decidir cuando se debe pasar de la ejecución de un hilo al siguiente.
- La operación de cambiar de hilo se llama cambio de contexto.
  - Un hilo puede ceder el control voluntariamente.
  - Un hilo puede ser desalojado por otro con prioridad mas alta.

¿Cómo trabajar con Hilos?

- La clase Thread y el interfaz Runnable se encuentran en: java.lang
- Puedo heredar de la clase Thread.
  - Ejemplo:
    - public class miHilo extends Thread { … }
- Implementando el interfaz Runnable.
  - Ejemplo:
    - public class miHijo implements Runnable { … }
- Desde un Applet (para hacer animaciones)
  - Ejemplo:
    - public class MyApplet extends Applet implements Runnable { .. }

La clase Thread y la interfaz Runnable

- Si utilizamos la interfaz **Runnable** tenemos que sobre escribir el método run().

- La clase **Thread** ofrece los siguientes métodos:
  - String getName() → Devuelve el nombre del hilo.
  - void setName(String nombre) → Establece el nombre del hilo.

  - int getPriority() → La prioridad de un hilo.
  - void setPriority(int prioridad) → Establece la prioridad.

  - boolean isAlive() → Devuelve true si el hilo está vivo.
  - void start() → Comienza un hilo llamando a su método run().
  - void run() → Punto de entrada de un hilo.
  - static void sleep(long milisg) → Suspende el hilo durante un tiempo.
  - void join() → Espera la terminación de un hilo.
  - String toString() → Devuelve el hilo como un String.
  - static Thread currentThread() → Devuelve el hilo principal de nuestro programa.
  - void yield() → Provoca hilos generosos, ceden el turno a otro hilo.

Constructores

- Cuando implemento el interfaz Runnable:
  - Thread(Runnable t);
  - Thread(Runnable t, String nombre);

- Cuando heredo:
  - Thread();
  - Thread(String nombre);

Ejemplo con Runnable

```
public class Prueba implements Runnable {
  Thread t; // Atributo de clase.

  public Prueba() {
    // Crea un segundo hilo:
    t = new Thread(this, "Hilo hijo creado");
    System.out.println("El hilo hijo es: " + t.toString());
    t.start(); // Empieza el hijo.
  }

  public void run() {
        // El método run hay que implementarlo.
    try {
      for(int i = 5; i > 0; i--) {
        System.out.println("El hijo escribe: " + i);
        Thread.sleep(500);
      }
    } catch (InterruptedException e) {
      System.out.println("Exception en el hilo hijo");
    }
    System.out.println("Muere el hijo.");
  }
}
```

```
public class Test {

  public static void main(String args[]) {
    new Prueba(); // Crea un nuevo thread
    try {

      for(int i = 5; i > 0; i--) {
        System.out.println("El hilo Padre escribe: " + i);
        Thread.sleep(1000);
      }
    } catch (InterruptedException e) {
      System.out.println("Exception en el Padre");
    }
    System.out.println("Muerte del padre");
  }
}
```

Ejemplo con la clase Thread

```
public class Prueba extends Thread {

  public Prueba() {
    // Crea el segundo hilo
        super( "Hilo hijo");
    System.out.println("El hilo hijo: " + this.toString());
    start(); // Inicio del hilo.
  }
  // This is the entry point for the second thread.
  public void run() {
    try {
      for(int i = 5; i > 0; i--) {
        System.out.println("El hijo escribe " + i);
        Thread.sleep(500);
      }
    } catch (InterruptedException e) {
      System.out.println("Exception en el hijo");
    }
    System.out.println("Muerte del hijo");
  }
}
```

```
public class Test {
  public static void main(String args[]) {
    new Prueba(); // Crea un nuevo hilo
    try {
      for(int i = 5; i > 0; i--) {
        System.out.println("El hilo Padre escribe: " + i);
        Thread.sleep(1000);
      }
    } catch (InterruptedException e) {
      System.out.println("Exception en Padre");
    }
    System.out.println("Muere el padre");
  }
}
```

El hilo principal / currentThread

- Cuando ejecutamos un programa java **de forma automática** se crea un hilo.
- Se denomina hilo principal.
- Se ejecuta al comenzar el programa.
  - A partir de este hilo se crean los demás.
  - Cuando este hilo finaliza, el programa principal termina.
- Se puede capturar con el siguiente método:
  - static Thread currentThread()

Ejemplo con currentThread

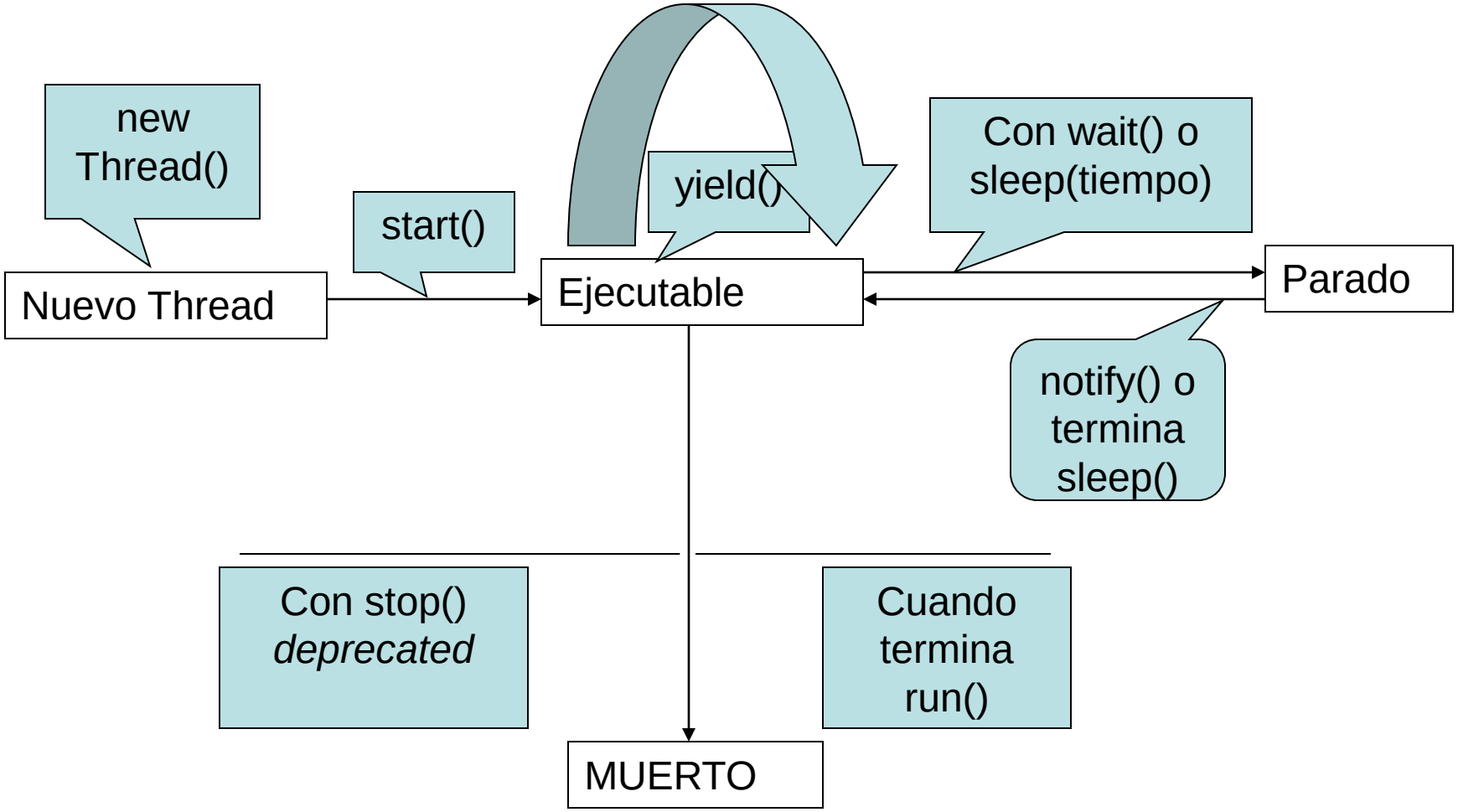
```
// Controlando el hilo principal.
public class CurrentThreadDemo {

public static void main(String args[]) {
    Thread t = Thread.currentThread();
    System.out.println("Current thread: " + t);
    // Cambiando el nombre del hilo
    t.setName("My Thread");
    System.out.println("After name change: " + t.toString());
    try {
      for(int n = 5; n > 0; n--) {
        System.out.println(n);
        Thread.sleep(1000);
      }
    } catch (InterruptedException e) {
      System.out.println("Main thread interrupted");
    }
  }
}
```

Constantes en la clase Thread

- Representan las prioridades de un hilo:
  - static int MAX_PRIORITY
  - static int MIN_PRIORITY
  - static int NORM_PRIORITY

- Ejemplo:
  - Thread t = new Thread();
  - t.setPriority(Thread.MAX_PRIORITY);

Ciclo de vida de un Hilo



Ciclo de vida de un Hilo

- Cuando creamos un hilo (… new Thread) colocamos al hilo en estado de nuevo hilo, pero todavía no ha arrancado.
- start() → Arranca el hilo, es decir, le manda a ejecución, método run().
- run() → Ejecución del hilo.
- sleep(tiempo) → El hilo se duerme el tiempo indicado, pasa a estado de parado. Cuando el tiempo de sleep pasa a ejecución.
- wait() → También para el hilo pero hasta que recibe notify() no se despierta y vuelve a ejecutarse.

Esperando a otros Hilos join()

- Podemos hacer a un hilo que espere a que termine otro hilo.

- Normalmente el hilo principal espera a que terminen los hilos hijos.

```
// Desde el hilo padre creamos un hijo y hacemos join(), el padre espera a que termine el hijo.
Thread t = new Thread("hilo1");
t.join();
```

Sincronización

- ¿Qué ocurre cuando dos o mas hilos tienen que acceder al mismo recurso?
- Se debe asegurar que sólo uno aceda al recurso en un instante dado.
- Este proceso recibe el nombre de sincronización.
- La clave es usar un monitor o semáforo.
- Uno consigue el recurso y cuando termina avisa a los demás.

Comunicación entre Threads wait() / notify()

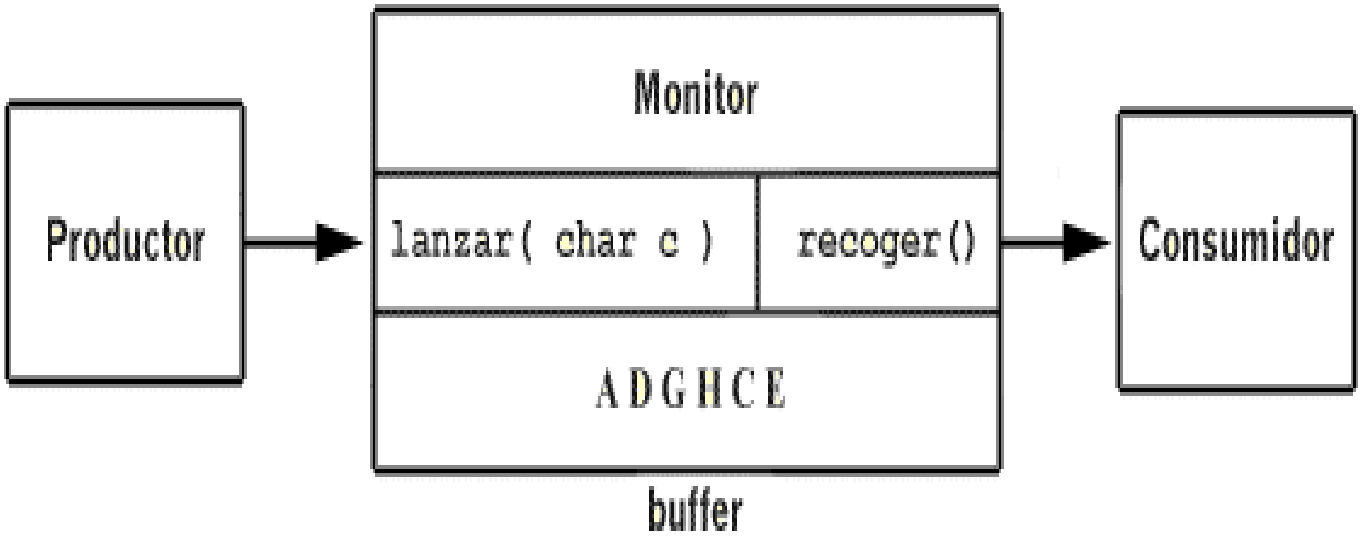
- El escenario:
  - Considerar un taxista y un pasajero como dos Threads independientes.
- El problema:
  - Cómo determinar cuando se ha llegado al destino.
- La solución:
  - El pasajero indica el destino al taxista donde va y pasa a la espera → **wait**();
  - El taxista avisa al pasajero cuando llegan al destino → **notify**();
- Ambos métodos están definidos en la clase **Object**

Métodos sincronizados

- Se trata de proteger las regiones críticas con métodos sincronizados.
- Una región crítica puede ser unos datos que comparten varios hilos.
- Sintaxis:

```
synchronized tipo_devuelto nombreMetodo(){
   // Instrucciones.
}
```

Productor / Consumidor



Problema del productor / consumidor

- El productor genera caracteres que coloca en un buffer (array) y el consumidor captura esos caracteres del buffer y los imprime por pantalla.
- La región crítica es el buffer, es compartida por ambos objetos.
  - Los métodos de lanzar y recoger serán sincronizados.

- El productor debe generar caracteres pero no debe superar el tamaño del buffer.
- El consumidor debe coger caracteres cuando haya caracteres dentro del buffer.

PRÁCTICAS: THREADS

Bloques synchronized

- A parte de tener métodos sincronizados que nos aseguran que ese método sólo lo va a ejecutar un Thread simultáneamente.

- También podemos tener bloques de código sincronizados dentro de un método.

Ejemplo

```
public class CriticalSection2 {
    public synchronized void someMethod_1() {
        // Este código sólo lo puede ejecutar un thread.
    }
    public void someMethod_11() {
        synchronized(this) {
            // Este código sólo lo puede ejecutar un thread
        }
    }
    public void someMethod_12() {
        // Múltiples Threads pueden ejecutar las instrucciones que se colocan aquí.
        synchronized(this) {
            // Este código sólo lo puede ejecutar un thread
        }
        // Múltiples Threads pueden ejecutar las instrucciones que se colocan aquí.
    }
}
```

- **synchronized(this) se puede sustituir por synchronized(CriticalSection2.class) esto lo utilizaríamos para los métodos static.**

Ejemplo

```
public static synchronized void someMethod_2() {
        // Este código sólo lo puede ejecutar un thread.
}
public static void someMethod_21() {
        synchronized(CriticalSection2.class) {
                // Este código sólo lo puede ejecutar un thread
        }
}
public static void someMethod_22() {
        // Múltiples Threads pueden ejecutar las instrucciones que se colocan aquí.
        synchronized(CriticalSection2.class) {
                // Este código sólo lo puede ejecutar un thread
        }
        // Múltiples Threads pueden ejecutar las instrucciones que se colocan aquí.
}
```

Uso de wait()

- El método **wait**() está en la clase **Object** y está declarado como **final**, NO se puede cambiar su comportamiento.
- Hay que llamarlo dentro de la zona sincronizada ya sea método o bloque.
- Y estará protegido por una condición, por ejemplo, un hilo consigue entrar en la región pero no hay sitio en el buffer o está vacío

Errores

- Una instancia de Object se puede utilizar para sincronizar:

```
public void wrongSynchronizationMethod {
    // objectRef se crea cada vez que se llama a este método.
    Object objectRef = new Object();

    synchronized(objectRef) {
    // No sincroniza porque cada hilo crea su objectRef …
    }
}
```

- ***El objeto que se utiliza para sincronizar tiene que ser COMÚN A TODOS LOS HILOS.***

notify / notifyAll

- notify despierta a un hilo, pero no hay forma de especificar a que hilo.
- La llamada a notify() elige un hilo de forma arbitraria.
- La llamada a notifyAll() despierta a todos los hilos, si hay dudas mejor utilizar notifyAll.

Código

```
public class WaitAndNotifyMethodCall {
        private Object objectRef = new Object();
        public synchronized void someMethod_1() {
                while (some condition is true) {
                        this.wait();
                }
                if (some other condition is true) {
                        // Notify all waiting threads
                        this.notifyAll();
                }
        }
        public static synchronized void someMethod_2() {
                while (some condition is true) {
                        WaitAndNotifyMethodCall.class.wait();
                }
                if (some other condition is true) {
                        // Notify all waiting threads
                        WaitAndNotifyMethodCall.class.notifyAll();
                }
        }

        public void someMethod_3() {
                synchronized(objectRef) {
                        while (some condition is true) {
                        objectRef.wait();
                        }
                        if (some other condition is true) {
                        // Notify all waiting threads
                        objectRef.notifyAll();
                        }
                }
        }
}
```

Errores

- Suponiendo el escenario de 1 productor y varios consumidores:

```
if (buffer is empty) {  // Si hay N consumidores → while
    buffer.wait();
}
buffer.consume();
```

- Si el buffer está vacío, todos los consumidores estarán esperando, si el productor produce un elemento y llama al método notifyAll TODOS los consumidores se despertarán pero sólo conseguirá 1 el bloqueo, y consumirá el elemento (buffer vacío), el siguiente consumidor con el código anterior se encontrará el buffer vacío y puede producir una exception.

- ***La llamada a consume tiene que estar después de una llamada a wait y protegida con una condición.***

notifyAll

- Cuando se llama a este método y hay varios hilos dormidos …
- ¿Qué hilo conseguirá el monitor para entrar en la RC?
- Depende de la planificación del sistema operativo.

¿Qué hilo se ejecuta?

- El método static **currentThread**() se puede utilizar para saber que hilo está en ejecución, en un método run() podemos hacer:

```
@Override
public void run() {
    Thread t = Thread.currentThread();
    String threadName = t.getName();
    System.out.println("Inside run() method: " + threadName);
}
```

Thread.sleep(milisg)

- El hilo se duerme unos milisegundos / nanosegundos.

- El método lanza InterruptedException:

```
try {
        System.out.println("I am going to sleep for 5 seconds.");
        Thread.sleep(5000); // The "main" thread will sleep
        System.out.println("I woke up.");
} catch(InterruptedException e) {
        System.out.println("Someone interrupted me in my sleep.");
}
```

Unidades de Tiempo

- La enum **TimeUnit** en el paquete **java.util.concurrent** representa varias unidades de tiempo: milisegundos, segundos …
  - TimeUnit.SECONDS.sleep(5);
  - Es lo mismo que Thread.sleep(5000);

Errores al esperar a que un hilo termine

```
public class JoinWrong {
    public static void main(String[] args) {
        Thread t1 = new Thread(JoinWrong::print);
        t1.start();
        System.out.println("We are done.");
        }

        public static void print() {
            for (int i = 1; i <= 5; i++) {
            try {
                System.out.println("Counter: " + i);
                Thread.sleep(1000);
            } catch (InterruptedException e) {
            e.printStackTrace();
            }
        }
    }
}
```

```
We are done.
Counter: 1
Counter: 2
Counter: 3
Counter: 4
Counter: 5
```

Join Correcto

```
Thread t1 = new Thread(JoinRight::print);
t1.start();
try {
    t1.join(); // "main" thread espera hasta que  t1 termine.
} catch (InterruptedException e) {
e.printStackTrace();
}
System.out.println("We are done.");
```

Ceder CPU

- Se puede ceder la ejecución a otros hilos:
  - Thread.yield()

Daemon

- Los hilos disponen del método **setDaemon(boolean)** y **isDaemon()**

- Cuando la propiedad se **establece a true**, y lanzamos un hilo daemon desde main, cuando termina main la JVM para también el hilo daemon.

- Si **daemon** es **false** aunque termine main el hilo **seguirá funcionando**.

Interrupt / Interrupted

- Un hilo también se puede interrumpir llamando al método **interrupt**() y se puede chequear si el hilo fue o no interrumpido (**interrupted**()).

- Podemos interrumpir al hilo:

  Thread.currentThread().**interrupt**();

Grupos de hilos

- La **JVM** crea un grupo de hilos llamado main y dentro de este grupo crea el hilo main que es el responsable de ejecutar el método **main**.

- También se puede crear grupos de hilos y agregar hilos a ese grupo.

- Los grupos representan estructuras jerárquicas en forma de árbol y se pueden anidar.

Grupos de hilos

- // Al crear el grupo se indica un identificador para el grupo:

  ThreadGroup myGroup = new ThreadGroup("My Thread Group");

- // Se pueden crear hilos que pertenezcan al grupo, indicando el grupo al crear el hilo.

  Thread t = new Thread(myGroup, "myThreadName");

- ***Se pueden aplicar acciones que afecten a todos los hilos del grupo:***
- https://docs.oracle.com/javase/8/docs/api/java/lang/ThreadGroup.html