# Hilos en java

Dora Angélica Ávila Galván

Roci Romero González

María Delia Sánchez Carmona

Alexsi Pérez Escudero

# Hilos (1/2)

- Los hilos permiten que el flujo del programa sea divido en dos o más partes, cada una ocupándose de alguna tarea de forma independiente.

# Hilos (2/2)

- Hay dos modos de conseguir threads en Java. Una es implementando la interface *Runnable*, la otra es extender la clase *Thread*.
- Heredar de Thread redefiniendo el método run().
- Crear una clase que implemente la interfaz Runnable que nos obliga a definir el método run().

# Clase Thread

- El primer método de crear un hilo es simplemente extender la clase Thread:

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

**Para instanciar**

```
Thread t = new MiHilo();
t.start();
```

# Interface Runnable

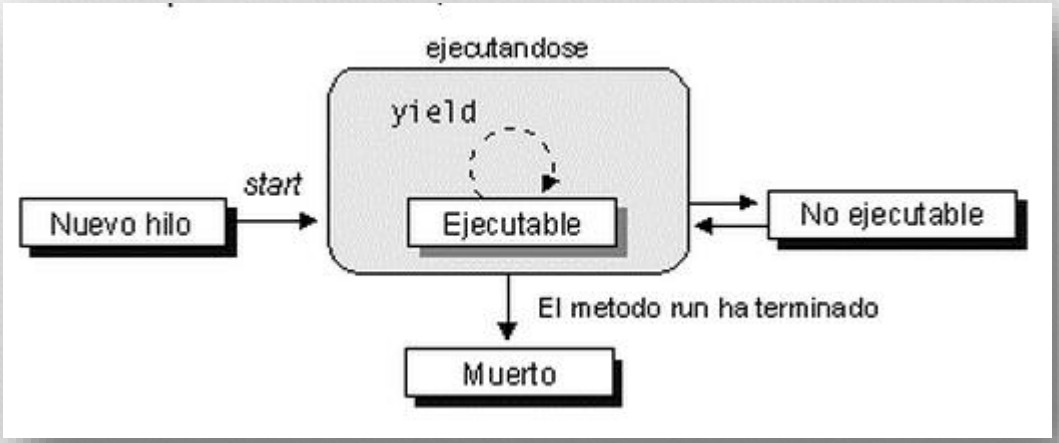
```
public class EjemploHilo implements Runnable
{
    public void run()
    {
        // Código del hilo
    }
}
```

**Para instanciar**

```
Thread t = new Thread(new EjemploHilo());
t.start();
```

# Ciclo de vida del hilo

- Cuando se invoca el método start() el hilo pasa a ser un hilo vivo, comenzándose a ejecutar el método run(), una vez que haya salido de este método pasará a su estado muerto.



# Synchronized (1/2)

- Se usa para indicar que ciertas partes del código (por lo regular, una función miembro) están sincronizadas, lo que quiere decir que solamente un subproceso puede acceder a dicho método a la vez.
- Permite asegurar el acceso competitivo a un objeto, se puede sincronizar métodos o bloques de sentencias.

# Synchronized (2/2)

- Asegura que mientras un hilo esté ejecutando un método (bloque) sincronizado de un objeto, ningún otro hilo podrá ejecutar un método (bloque) sincronizado del mismo objeto.

# Ejemplo (1/3)

```
class Cola {
        private int [] datos;
        private int entrada, salida, ocupados, tamano;
        public Cola (int tamanio) {
                datos = new int [tamanio];
                nuevotamanio = tamanio;
                ocupados = 0;
                banderaEntrada = 1;
                banderaSalida  = 1;
        }
public synchronized void almacena (int x) {…}
public synchronized int obtener () { … }
}
```

**Antes de realizar la operación se va a asegurar de que no hay otro hilo usando el objeto Cola**

# Ejemplo (2/3)

```
public synchronized void almacena (int x) {
        try {
                while (ocupado == tamanio) wait();
                datos [banderaEntrada] = x;
                banderaEntrada = (banderaEntrada +1 ) % tamanio;
                ocupado ++;
                notify ();
        }
        catch (InterruptedException e) {}
}
```

# Ejemplo (3/3)

```
public synchronized int obtener () {
        int x = 0;
        try {
                while (ocupado == 0 ) wait ();
                x = datos [banderaSalida];
                banderaEntrada = (banderaSalida + 1)% tamanio;
                ocupado--;
                notify ();
        }
        catch ( InterruptedException e) {}
        return x;
}
```

# Join

- El método join () permite a un hilo quedar a la espera a que termine un segundo hilo, por lo cual, se utiliza para mantener un orden en la secuencia de los hilos. Este método debe controlarse mediante la excepción InterruptedException para evitar errores de compilación.

# Ejemplo

```
public class jointest {
        public static void main (String [] args){
                Vector a, b;
                a=new Vector (50, “a”);
                b=new Vector (100, “b”);
                a.start();
                b.start();
                try {
                        a.join ();
                        b.join ();
                }
                catch (InterruptedException e){}
                System.out.println (“Suma (x) a:” + a.suma());
                System.out.println (“Suma (x) b:” + b.suma());
                System.out.println (“Suma (x^2) a:” + a.sumaSqr());
                System.out.println (“Suma (x^2) b:” + b.sumaSqr());
                System.out.println (“Media a :” + a.media());
                System.out.println (“Media b :” + b.media());
        }
}
```

# Sleep

- El método sleep() sirve para indicarle al hilo que se duerma, es necesario especificar el tiempo (milisegundo). Es utilizado cuando se pretende retrasar la ejecución del hilo.

- Sleep() no consume recursos del sistema mientras el hilo duerme. De esta forma otros hilos pueden seguir funcionando.

# Ejemplo

```
Thread  Consumidor = new Claseconsumidor();
Consumidor.start();
try {
    Consumidor.sleep( 10000 );
} catch( InterruptedException e ) {
    ;
    }
```

# Notify y notify all

- Para lograr una buena sincronización entre las tareas, se debe hacer uso de otros mecanismos de sincronización.
- Para lograr alternancia, usamos los métodos llamados wait y notify.
- El método notify() sólo despierta o desbloquea un hilo, si lo hay esperando. En cambio, notifyAll() despierta a todos los que estén esperando.

# Wait (1/2)

- El método wait() hará que el hilo que invoca se bloquee hasta que ocurra un *timeout* u otro hilo llame al método notify() o notifyAll() sobre el mismo objeto (lo primero que ocurra).

- Cuando un hilo llama a wait(), la llave que éste tiene es liberada, así otro proceso que esperaba por ingresar al monitor puede hacerlo.

# Wait (2/2)

- Luego que un hilo despierta y como parte del wait() tratará de reingresar al monitor pidiendo la llave nuevamente, podría tener que esperar a que otro hilo la libere.

- Los llamados wait(), notify() y notifyAll(), sólo pueden ser llamados dentro de un método o bloque sincronizado.

```
import java.io.*;
public class PandC {
  static int produceSpeed = 200;
  static int consumeSpeed = 200;
  public static void main (String args[]) {
    if (args.length > 0)
      produceSpeed = Integer.parseInt (args[0]);
    if (args.length > 1)
      consumeSpeed = Integer.parseInt (args[1]);
    Monitor monitor = new Monitor();
    new Producer(monitor, produceSpeed);
    new Consumer(monitor, consumeSpeed);
    try {
      Thread.sleep(10000);
    } catch (InterruptedException e) {
    }
    System.exit(0);
  }
}
```

```
class Monitor {
  PrintWriter out = new PrintWriter (System.out, true);
  int token;
  boolean valueSet = false;
    //get token value
  synchronized int get () {
    if (! valueSet)
      try {
        wait();
      } catch (InterruptedException e) {
      }
    valueSet = false;
    out.println ("Got: " + token);
    notify();
    return token;
  }
synchronized void set (int value) {//set token value
    if (valueSet)
      try {
        wait();
      } catch (InterruptedException e) {
      }
    valueSet = true;
    token = value;
    out.println ("Set: " + token);
    notify();
  }
}
```

```
class Producer implements Runnable {
  Monitor monitor;
  int speed;
  Producer (Monitor monitor, int speed) {
    this.monitor = monitor;
    this.speed = speed;
    new Thread (this, "Producer").start();
  }
  public void run() {
    int i = 0;
    while (true) {
      monitor.set (i++);
      try {
        Thread.sleep ((int) (Math.random() * speed));
      } catch (InterruptedException e) {
      }
    }
  }
}
```

```
class Consumer implements Runnable {
  Monitor monitor;
  int speed;
  Consumer (Monitor monitor, int speed) {
    this.monitor = monitor;
    this.speed = speed;
    new Thread (this, "Consumer").start();
  }
  public void run() {
    while (true) {
      monitor.get();
      try {
        Thread.sleep ((int) (Math.random() * speed));
      } catch (InterruptedException e) {
      }
    }
  }
}
```