T2.5. Creación, inicio, prioridad, sincronización y finalización de hilos

Creación, inicio y finalización.

- Podemos heredar de Thread o implementar `Runnable`. Usaremos el segundo recordando implementar el método `public void run()`.
- Para crear un hilo asociado a un objeto usaremos algo como:

```
Thread hilo=new Thread(objetoDeClase)
```

Lo más habitual es guardar en un vector todos los hilos que hagan algo, y no en un objeto suelto.

- Cada objeto que tenga un hilo asociado debe iniciarse así:

```
hilo.start();
```

- Todo programa multihilo tiene un «hilo principal», el cual deberá esperar a que terminen los hilos asociados ejecutando el método `join()` .

Sincronización de hilos.

Cuando un método acceda a una variable miembro que esté compartida deberemos proteger dicha sección crítica, usando `synchronized`. Se puede poner todo el método `synchronized` o marcar un trozo de código más pequeño.

Prioridades de los hilos.

Podemos asignar distintas prioridades a los hilos usando los campos estáticos `MAX_PRIORITY` y `MIN_PRIORITY`. Usando valores entre estas dos constantes podemos hacer que un hilo reciba más procesador que otro (se hace en contadas ocasiones).

Para ello se usa el método `setPriority(valor)`

Gestión de prioridades.

En realidad un sistema operativo no está obligado a respetar las prioridades, sino que se lo tomará como «recomendaciones». En general hasta ahora todos respetan hasta cierto punto las prioridades que pone el programador pero no debe tomarse como algo absoluto.

Programación de aplicaciones multihilo.

La estructura típica de un programa multihilo es esta:

```
class TareaCompleja implements Runnable{
        @Override
        public void run() {
                for (int i=0; i<100;i++){
                        int a=i*3;
                }
                Thread hiloActual=Thread.currentThread();
                String miNombre=hiloActual.getName();
                System.out.println(
                                "Finalizado el hilo"+miNombre);
        }
```

```
}
public class LanzadorHilos {
	public static void main(String[] args) {
		int NUM_HILOS=100;
		Thread[] hilosAsociados;

		hilosAsociados=new Thread[NUM_HILOS];
		for (int i=0;i<NUM_HILOS;i++){
			TareaCompleja t=new TareaCompleja();
			Thread hilo=new Thread(t);
			hilo.setName("Hilo: "+i);
			hilo.start();
			hilosAsociados[i]=hilo;
		}

		/* Despues de crear todo, nos aseguramos
		 * de esperar que todos los hilos acaben. */

		for (int i=0; i<NUM_HILOS; i++){
			Thread hilo=hilosAsociados[i];
			try {
				//Espera a que el hilo acabe
				hilo.join();
			} catch (InterruptedException e) {
				System.out.print("Algun hilo acabó ");
				System.out.println(" antes de tiempo!");
			}
		}
		System.out.println("El principal ha terminado");
	}
}
```

Supongamos que la tarea es más compleja y que el bucle se ejecuta un número al azar de veces. Esto significaría que nuestro bucle es algo como esto:

```
Random generador= new Random();
int numAzar=(1+generador.nextInt(5))*100;
for (int i=0; i<numAzar;i++){
	int a=i*3;
}
```

¿Como podríamos modificar el programa para que podamos saber cuantas multiplicaciones se han hecho en total entre todos los hilos?

Aquí entra el problema de la sincronización. Supongamos una clase contador muy simple como esta:

```
class Contador{
	int cuenta;
	public Contador(){
		cuenta=0;
	}
	public void incrementar(){
		cuenta=cuenta+1;
	}
	public int getCuenta(){
		return cuenta;
	}
}
```

De esta forma podríamos construir un objeto contador y pasárselo a todos los hilos para que en ese único objeto se almacene el recuento final. El problema es que en la programación multihilo **SI EL OBJETO CONTADOR SE COMPARTE ENTRE VARIOS HILOS LA CUENTA FINAL RESULTANTE ES MUY POSIBLE QUE ESTÉ MAL**

Esta clase debería tener protegidas sus secciones críticas

```
class Contador{
	int cuenta;
	public Contador(){
```

```
                cuenta=0;
        }
        public synchronized void incrementar(){
                cuenta=cuenta+1;
        }
        public synchronized int getCuenta(){
                return cuenta;
        }
}
```

Se puede aprovechar todavía más rendimiento si en un método marcamos como sección crítica (o sincronizada) exclusivamente el código peligroso:

```
public  void incrementar(){
        System.out.println("Otras cosas");
        synchronized(this){
                cuenta=cuenta+1;
        }
        System.out.println("Mas cosas...");
        synchronized(this){
                if (cuenta>300){
                        System.out.println("Este hilo trabaja mucho");
                }
        }
}
```