10. Hilos

- Definición y ejecución
  - `Thread` y `Runnable`
- Ciclo de vida de un hilo
  - Interrupción
  - Bloqueo:
    - `sleep`
    - Sincronización

Hilos

- Trasladan el concepto de multitarea a la ejecución de un programa.
- Resuelven problemas de difícil solución sin ellos:
  - ¿Cómo conseguimos atender la pulsación de teclas del usuario y conceder el turno a las entidades del juego para que se muevan independientemente?
- **Hilo = flujo de ejecución**
- El uso de hilos conlleva muchos **RIESGOS**:
  - Distintos flujos de ejecución accediendo a los mismos objetos del programa (!!!)
  - ➔Surgen todos los problemas de la programación concurrente

Definición y Ejecución de un hilo

- Un hilo es un objeto cuya clase hereda de **`Thread`**
- Define su flujo de ejecución en un método **`run()`**
- Ejemplo:

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

- Comienzan su ejecución con una llamada a **`start`**`()`

```
Temporizador temp = new Temporizador();
temp.start();
```


Problema con la herencia

- Heredar de `Thread` puede resultar inconveniente: herencia simple.
- Solución: uso de **interfaces**

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

- El hilo tiene la opción de implementar la interfaz:

```
public class Temporizador extends Notificador
                          implements Runnable{
   public void run() { ...}
```

- Pero la clase **no es un hilo**, hace falta convertirla:

```
Temporizador temp = new Temporizador();
Thread hilo = new Thread(temp);
hilo.start();
```

- **Todas las operaciones sobre el hilo hay que hacerlas con la variable `Thread`.**

Finalización de un Hilo

- El flujo principal de un **programa** acaba cuando termina el método `main`.
- Un hilo acaba cuando finaliza el método `run()` normal o por una excepción no capturada.
- A veces es necesario **detener un hilo**, pero debido a cuestiones de concurrencia la detención debe ser **ordenada: Interrupción**

```
hilo.interrupt();
```

- El hilo debe controlar cuando se solicita su interrupción:

```
if (Thread.interrupted()) ...
```

- Consideraciones método **interrupted()**:
  - El método `interrupted` es de **clase** para poder llamarlo desde una clase que no herede de `Thread`
  - La llamada al método resetea un *flag de interrupción*. Alternativa:

```
Thread.currentThread().isInterrupted();
```


Ciclo de vida

- Un hilo puede estar en cuatro estados:
  - **Inicial**: antes de ejecutar `start()`. En realidad aún no es un hilo.
  - **En ejecución**: tras ejecutar `start()` y durante el método `run()`.
  - **Bloqueado**:
    - En un candado (**sincronización** ...)
    - **Durmiendo**: `Thread.sleep(milisegundos);`
  - **Finalizado**.
- Podemos averiguar si el hilo está en ***ejecución*** **o b*loqueado***:

```
hilo.isAlive();
```

Método `sleep`

- Un hilo puede ser interrumpido durante el sueño con una llamada `interrupt()`.
- El método `sleep` lanza la excepción `InterruptedException`

```
while (true) {
  try {
   Thread.sleep(1000);
   System.out.println("hola");
  } catch (InterruptedException e) { ...}
}
```

- Cuando un hilo es interrumpido durante el sueño no queda activado el flag de interrupción (!!)

Sincronización

- Varios hilos accediendo a los mismos datos pueden dar lugar a **inconsistencias**.
- Ejemplo: clase `Punto` método `incX()`
  - `x = x + 1;`
- Hay que contemplar la posibilidad de que un método sea ejecutado concurrentemente y protegerlo: (**esquema a priori**)
  - `public synchronized void incX() { ... }`
  - La llamada al método bloquea al hilo si hay otro ejecutando esa operación sobre el mismo objeto.
- Si la clase no está protegida, los hilos deben evitar los problemas de concurrencia poniendo un candado sobre el objeto: (**esquema a posteriori**)

```
synchronized(pto) {
    pto.incX();
    ...
}
```

Cuestiones avanzadas y Conclusiones

- Grupos de hilos.
- Prioridades.
- Estrategias de sincronización
- ...

- **Consejos**:
  - Programar hilos puede ser un perjuicio más que un beneficio.
  - **Hay que utilizar los hilos con sensatez**

Ejemplo - Temporizador

```
import java.util.*;

public class Temporizador extends Thread {

   private int espera;

   private List despertables = new ArrayList();

   public Temporizador (int espera) {
        this.espera = espera;
   }
   public void addDespertable (Despertable d) {
        despertables.add(d);
   }
   public void removeDespertable (Despertable d) {
        despertables.remove(d);
   }
```

Ejemplo - Temporizador

```
   public void run () {
        while (!interrupted()) {
               try {
                       Thread.sleep(espera);
               } catch (InterruptedException e) {
                       break;
               }
               Iterator it = despertables.iterator();
               while (it.hasNext()) {
                 Despertable d = (Despertable)it.next();
                 d.despierta();
               }
        }
   }
}
```

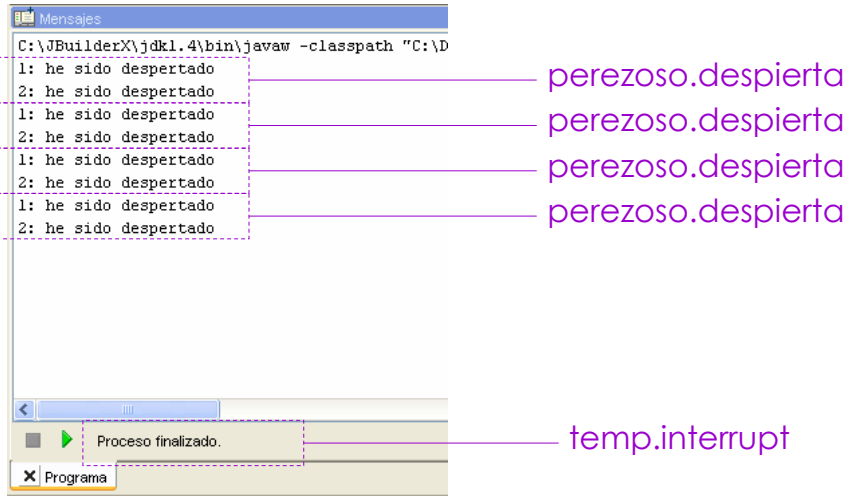

Ejemplo - Temporizador

```
public interface Despertable {
   void despierta();
}

public class Perezoso implements Despertable {

   private String nombre;

   public Perezoso (String nombre) {
        this.nombre = nombre;
   }

   public void despierta () {

        System.out.println(nombre + ": he sido despertado");
   }
}
```

Ejemplo - Programa

```
public class Programa {
   public static void main (String[] args) {
        Temporizador temp = new Temporizador (1000);

        temp.addDespertable(new Perezoso("1"));
        temp.addDespertable(new Perezoso("2"));

        temp.start();

        //Esperamos 5 sg para interrumpir la alarma
        try {
                Thread.sleep(5000);
        } catch (InterruptedException e) {}

        temp.interrupt();

        // Espero a que termine
        while (temp.isAlive());
   }
}
```


Ejemplo- salida del programa



Reflexión en Java

- Paquete **`java.lang.reflect`**
- Incluye las clases:
  - `Field`
  - `Method`
  - `Constructor`
- Permite:
  - Examinar las propiedades de una clase en tiempo de ejecución.
  - Inspeccionar los objetos en tiempo de ejecución
  - Utilizar los objetos `Method` como si fueran punteros a función

clase `Class`

```
• public static Class forName(String className)
• Field [] getFields()
• Field [] getDeclaredFields()
• Field getField(String name)
• Method [] getMethods
• Method [] getDeclaredMethods()
• Method getMethod(String name,
                   Class[] parameterTypes)
• Constructor [] getConstructors
• Constructor [] getDeclaredConstructors()
• String getName()
• boolean isInstance(Object obj) //equivale instanceof
```

clases Field y Method

- Clase **Field**:
  - `public Object get(Object obj)`
  - `public Class getType()`
  - `public String getName()`

- Clase **Method**:
  - `public Class[] getExceptionTypes()`
  - `public Class[] getParameterTypes()`
  - `public Class getReturnType()`
  - `public String getName()`
  - `public Object invoke(Object obj, Object[] args)`
    - si obj = null invocamos un método de clase

Ejemplo: invocar método de clase

```
class Cuenta{
  public static Cuenta abrir(Persona titular){
  …
   }
}
```

- Invocar al método de clase mediante reflexión:

```
Persona titu = bd.getCliente("pepito");
Class claseCta = Class.forName("banco.Cuenta");
Method metodoAbrir = claseCta.getMethod("abrir",
                                  new Class[]{Persona.class});
Cuenta cta = (Cuenta)metodoAbrir.invoke(null,
                                        new Object[] {titu});
```