Anexo II .- Principales interfaces

Las interfaces, como hemos visto en la teoría, nos permiten declarar qué queremos sin necesidad de saber en ese momento los detalles de la implementación. Además, cuando trabajamos en grupo, también nos permite establecer unas reglas que todos debemos cumplir.
El lenguaje Java tiene varias interfaces predefinidas. Echemos un vistazo a las más interesantes.

Collection

Aunque ya las hemos visto en la teoría, recordemos que representen un grupo de objetos. Cada uno de estos objetos se denomina elemento. Cuando queremos trabajar con un conjunto de elementos, necesitamos un almacén donde poder guardarlos, y este almacén es precisamente lo que es la interfaz **Collection**. Gracias a esta interfaz, podemos guardar cualquier tipo de objeto y utilizar una serie de métodos comunes.
A partir de esta interfaz se extienden otra serie de interfaces genéricas que añaden diferentes funcionalidades como son:

- **Set**. Los elementos no pueden repetirse. Ejemplos de implementaciones de Set son: HashSet, TreeSet y LinkedHashSet.
- **List**. Una succesión de elementos, que pueden repetirse, a los cuales se puede acceder posicionalmente. Ejemplos de implementaciones de List son: ArrayList y LinkedList.
- **Map**. Asocia claves con valor. No puede haber claves duplicadas y cada clave puede tener como máximo un valor asociado. Ejemplos de implementaciones de Map son: HashMap, TreeMap y LinkedHashMap.

Iterator

Esta interfaz provee un mecanismo estándar para acceder secuencialmente a los elementos de una coleción. Se trata de un patrón estándar de diseño, empleado en muchos lenguajes de programación, que tiene como finalidad poder acceder a los contenidos de los objetos incluidos sin exponer su estructura.
La interfaz **Iterator** obliga a implementar en las clases que la implementan los siguientes métodos:

- `boolean hasNext()` devuelve true si la iteración tiene más elementos.
- `E next()` devuelve el siguiente elemento de la iteración.
- `void remove()` elimina de la colección el último elemento devuelto por `next()`.

Con los iteradores podemos construir diferentes tipos de bucles:

```
for (Iterator<E> ite = ...; ite.hasNext(); ) {
      E elemento = ite.next();
      ...
}
```

```
Iterator<E> ite = ...;
while (ite.hasNext()) {
```

```
      E elemento = ite.next();
      ...
}
```

```
X objeto = ...;
for(E e: objeto) {
      ...
}
```

La mayoría de colecciones de Java implementan esta interfaz y por tanto se pueden iterar obteniendo la Iterator mediante el método `iterator()`.

Vemos un ejemplo de implementación de Iterator. Un generador de números enteros aleatorios mediante Iterator:

```
public class SerieAleatoria implements Iterator<Integer> {
    private Random random;

    public SerieAleatoria() {
        random = new Random();
    }

    public boolean hasNext() {
        return true;
    }

    public Integer next() {
        return random.nextInt();
    }

    public void remove() {
        throw new UnsupportedOperationException();
    }
}
```

Iterable

Esta interfaz permite especificar en las clases que la implementan, que los elementos que contienen se pueden iterar mediante un **Iterator**.

Las clases que implementan esta interfaz tienen que proporcionar un método denominado iterator() que devuelva un **Iterator**.

- `Iterator<T> iterator();`

¿Cómo podemos obtener un iterator? Veamos algunos ejemplos:

List

```
List<T> lista = new ArrayList<T>();
. . .
Iterator<T> ite = lista.iterator();
while (ite.hasNext()) {
```

```
      T elemento = itr.next();
      . . .
}
```

Set

```
Set<T> conjunto = new HashSet<T>();
. . .
Iterator<T> ite = conjunto.iterator();
while (ite.hasNext()) {
      T elemento = itr.next();
      . . .
}
```

Si tenemos un array, podemos pasarlo a lista e iterar sobre la lista de la siguiente forma:

```
T[] array
List<T> lista = Arrays.asList(array);
Iterator<T> ite = lista.iterator();
…
```

Comparable

Esta interfaz es útil para indicar las clases que tienen una relación de orden total, lo que se traduce en el hecho que las clases que implementan esta interfaz tienen que proporcionar el método:

- `int compareTo(T x)` compara *this* con x, devolviendo:
  - un número negativo (habitualmente -1) si *this* < x
  - cero si *this* == x
  - un número positivo (habitualmente 1) si *this* > x

Una gran ventaja de implementar esta interfaz es que podemos utilizar el método sort() para ordenar los elementos de una colección.

Vemos un ejemplo:

```
public class Persona implements Comparable<Persona> {
      private String numero;
      private int edad;
      ...
      public Persona(String numero, int edad) {
            this.número = numero;
            this.edad = edad;
      }
      ...
      @Override
```

```
	public int compareTo(Persona p) {
		return this.numero.compareTo(p.getNombre());
	}
}
```

de esta forma si ferem:

```
ArrayList<Persona> personas = new ArrayList<>();
personas.add(new Persona("Mario", 19));
personas.add(new Persona("Fernando", 24));
personas.add(new Persona("Omar", 27));
personas.add(new Persona("Juana", 22));
System.out.println(personas);
```

Mostraría

[Persona{numero='Mario',edad=19}, Persona{numero='Fernando',edad=24}, Persona{numero='Omar', edad=27}, Persona{numero='Juana',edad=22}]

Si ahora ferem:

```
Collections.sort(personas);
System.out.println(personas);
```

Mostraría

[Persona{numero='Fernando',edad=24}, Persona{numero='Juana',edad=22}, Persona{numero='Mario', edad=19}, Persona{numero='Omar',edad=27}]

Cómo podemos observar con poco esfuerzo, y sobre todo de manera uniforme, podemos ordenar los elementos de nuestras listas.

Comparator

Esta interfaz es similar al anterior, pero en lugar de comparar el valor propio con el recibido como parámetro, compara los dos valores recibidos como parámetros. Las clases que implementan esta interfaz deben proporcionar el método:

- `int compare(T x1, T x2)` compara x1 con x2, devolviendo:
  - un número negativo (habitualmente -1) si x1 < x2
  - cero si x1 == x2
  - un número positivo (habitualmente 1) si x1 > x2

Continuando con el ejemplo anterior, imaginemos que no queremos ordenar las personas por nombre sino por edad, ¿cómo lo haríamos? **Comparator** tiene la respuesta.

Para poder tener diferentes formas de ordenar los elementos de una colección podemos definir varias clases que implementen Comparator, cada una de ellas ordenará por un atributo. En este ejemplo, para ordenar por edad, crearíamos una clase que implemente el Comparator:

```
public class OrdenaPersonaPorEdad implements Comparator<Persona> {
```

```
        @Override
        public int compare(Persona o1, Persona o2) {
            return o1.getEdad() - o2.getEdad();
        }
}
```

y después utilizaríamos esa clase para ordenar por edad:

```
personas.sort(new OrdenaPersonaPorEdad());
System.out.println(personas);
```

Mostraría:
[Persona{numero='Mario',edad=19}, Persona{numero='Juana',edad=22}, Persona{numero='Fernando', edad=24}, Persona{numero='Omar',edad=27}]

Si queremos evitar tener que crear un archivo para cada una de las clases de los campos que queremos ordenar, podemos hacerlo mediante una clase estática dentro de Persona, de esa forma como pertenece a la clase Persona, queda más ordenado.

```
public class Persona implements Comparable<Persona> {
      ...
      public static class ComparatorEdad implements Comparator<Persona> {
        @Override
        public int compare(Persona o1, Persona o2) {
            return o1.getEdad() - o2.getEdad();
        }
    }
}
```

y para ordenar:

```
personas.suerte(new Persona.ComparatorEdad());
System.out.println(personas);
```

Cloneable

En algunas situaciones deseamos que un objeto que se pasa a una función/método no se modifique en el curso de la invocación. En estos casos, puede ser muy útil para el programador realizar una copia del objeto original y realizar las transformaciones en la copia dejando intacto el original.
La interfaz **Cloneable** está creada con esa intención. No define ningún método pero las clases que la implementan, indican al método *clone()* de la clase base Object, que se pueden hacer copias miembro a miembro de las instancias de la clase. Por lo tanto, cualquier método que implemente Cloneable tiene que sobrescribir el método:

- `Object clone();`

Si una clase no implementa esta interfaz, y se intenta hacer una duplicación mediante clone, dará como

resultado una excepción del tipo *CloneNotSupportedException*.

Vemos un ejemplo de uso:

```
public class Punto implements Cloneable {
	private int x;
	private int y;
	...
	@Override
	public Object clone() {
		Object obj = null;
		try {
			obj = (Punto)super.clone();
		} catch (CloneNotSupportedException cnse) {
			System.out.println("Clonación no soportada");
		}
		return obj;
	}
}
```

de este modo si ahora ejecutásemos el método clone() sobre un objeto de tipo Punto, haríamos una copia exacta de él.

```
Punto p1 = new Punto(2,5);
Punto p2 = p1.clone();
```

Ahora p1 y p2 son dos objetos distintos pero que tienen los mismos valores.

El ejemplo que acabamos de ver está bien, pero, ¿qué pasaría si lo intentásemos en una clase compuesta?, es decir, en una clase que tenga como atributos otras clases. La respuesta es que no funcionaría correctamente, solo clonaría el objeto simple. Se debe implementar Cloneable en toda la jerarquía de clases de las cuales esté compuesta la clase que queramos clonar.
Vemos un ejemplo:

```
public class Rectangulo implements Cloneable {
	private int ancho ;
	private int alto ;
	private Punto origen;
	...
	public Object clone(){
		Rectangulo obj=null;
		try{
			obj=(Rectangulo)super.clone();
		} catch(CloneNotSupportedException ex){
			System.out.println("Clonación no soportada");
		}
```

```
            obj.origen=(Punto)obj.origen.clone();
            return obj;
    }
}
```

Cómo podemos observar en el ejemplo, clonamos en primer lugar el objeto Rectangulo y si todo va bien, volvemos a clonar el atributo origen, que al ser de tipo Punto implementa la interfaz clonable y por tanto podemos ejecutar de nuevo el método clone.

Clonable es un intento por parte de Java de ofrecer un mecanismo de copia, pero la realidad es que en la práctica tiene poco de uso por el esfuerzo que supone la clonación de objetos complejos y que hay alternativas mejores a la clonación, como por ejemplo la libreria Apache-commons (SerializationUtils y BeanUtils) y otras que veremos en la Serialización de objetos.
Otra alternativa a la clonación es definir un constructor que reciba como parámetro un objeto de la clase y el copie sobre el objeto creado.

Serializable

Una característica del lenguaje Java es la Serialización de objetos. Esta característica permite convertir cualquier objeto que implemente la interfaz **Serializable**, en una secuencia de bytes que pueden ser posteriormente leídos para restaurar el estado original.
La serialización, unida a la persistencia de datos (mediante archivos o bases de datos), permite a las aplicaciones guardar el estado completo de la aplicación en un determinado momento y poder recuperarlo más tarde. Es más, permite guardar el estado completo, enviarlo a través de una red y recuperarlo en otra máquina.
La interfaz Serializable es una interfaz vacía, es decir, no define ningún método. Para implementarla simplemente tenemos que añadir `implements Serializable` para informar en Java, más concretamente en la clase `ObjectOutputStream`, que puede utilizar el método `writeObject` para serializar nuestra clase.

Para leer (deserializar) los objetos guardados previamente, hacemos uso del método readObject de la clase ObjectInputStream. Este método lee un flujo de entrada y reconstruye los objetos de la clase serializada.
Vemos un ejemplo (asumiendo que la clase Lista implementa la interfaz Serializable):

```
Lista lista1= new Lista(new int[]{12, 15, 11, 4, 32});
FileOutputStream fileOut=new FileOutputStream("media.obj");
ObjectOutputStream salida=new ObjectOutputStream(fileOut);
salida.writeObject(lista1);
salida.close();
```

Para recuperar los datos serializados haríamos lo siguiente:

```
FileInputStream fileIn = new FileInputStream("media.obj");
ObjectInputStream entrada=new ObjectInputStream(fileIn);
Lista obj1=(Lista)entrada.readObject();
entrada.close();
```

Cuando un atributo de una clase contiene información sensible, hay disponibles varias técnicas para protegerla. Incluso cuando la información es privada (el atributo tiene el modificador private) una vez que se ha enviado al flujo de salida alguien puede leerla en el archivo en disco o interceptarla en la red.
La manera más simple de proteger la información sensible, como una contraseña (password) es la utilizar el modificador ***transient*** antes del atributo que la guarda.

Serialización personalizada
El proceso de serialización proporcionado por el lenguaje Java, suele ser suficiente para la mayor parte de las clases, ahora bien, se puede personalizar para aquellos casos específicos.
Para personalizar la serialización, es necesario reescribir los métodos writeObject y readObject. El primero controla qué información es enviada al flujo de salida. El segundo, lee la información escrita por writeObject.

```
private void writeObject (ObjectOutputStream s) throws IOException{
      s.defaultWriteObject();
      //...código personalizado para escribir los datos
}
private void readObject (ObjectInputStream s) throws IOException{
      s.defaultReadObject();
      //...código personalizado para leer los datos
}
```

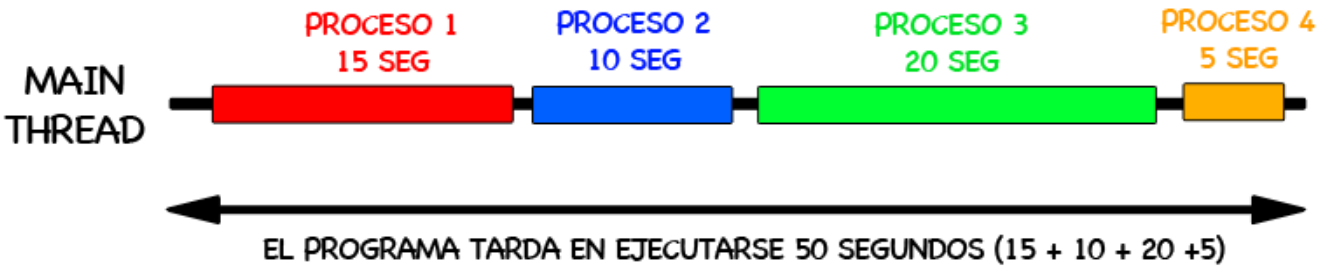
Cuando veamos la gestión de ficheros volveremos a hablar sobre el tema y lo veremos con más detalle, puesto que hay instrucciones para trabajar en ficheros que, hoy por hoy, desconocemos.

Para acabar, hay que decir que aunque la serialización de objetos proporcionada por el lenguaje ha sido ampliamente utilizada a lo largo de muchos años, es cierto que cada vez se utiliza menos por el hecho que han aparecido alternativas más rápidas, con menor consumo de memoria y más transportables como son los formatos JSON y XML.
Un ejemplo de librerías que utilizan JSON para serializar objetos son:

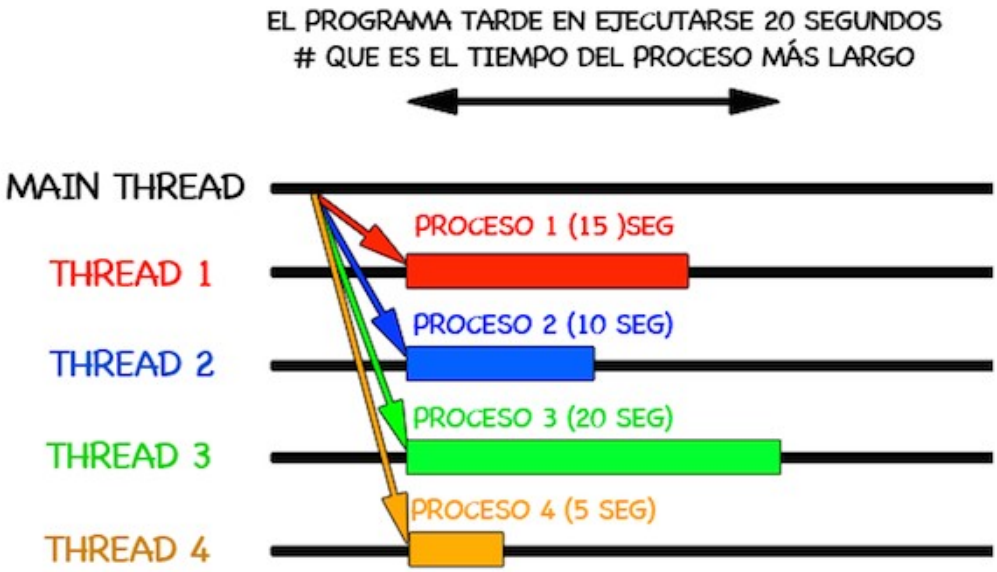
- Gson
- Jackson

Runnable

La multitarea es la capacidad que tiene un sistema de poder ejecutar varios procesos a la vez, es decir, concurrentemente. El número de procesos que el sistema podrá ejecutar de forma concurrente dependerá del número de CPU's y del número de núcleos que tenga cada CPU.
En Java podemos diseñar nuestros programas para que puedan hacer uso de la multitarea.
En la siguiente imagen podemos ver 4 procesos ejecutándose de forma secuencial:



Este proceso pesado tarda 50 segundos en realizar su tarea, que es la suma de lo que tardan cada uno de los procesos ligeros.
Si en lugar de ejecutarlos de forma secuencial, los ejecutásemos en paralelo, reduciríamos de forma considerable el tiempo total de ejecución.



En Java podemos implementar la multitarea utilizando la clase **Thread** (es decir que la clase que implementamos debe extender (heredar de) la clase Thread) y la clase Thread implementa la Interface `Runnable`. En el siguiente diagrama de clase se muestra la Interface Runnable y la clase Thread con sus principales métodos:

Por lo tanto, podemos implementar la multitarea creando una subclase de Thread y también lo podemos hacer implementando la interfaz Runnable.
Cuando implementamos la interfaz Runnable, debemos implementar el método `run()` que es el que realizará toda la tarea del hilo de ejecución.
Más adelante, en el tema de hilos de ejecución lo veremos en detalle.

Callable

Hemos visto que la interfaz Runnable permite crear hilos de ejecución implementando el método `run()`, pero uno de los principales problemas que tiene hacerlo de esta forma, es que no devuelve ningún resultado. Si necesitamos ejecutar una tarea en paralelo y que nos devuelva un resultado en el futuro, podemos hacer uso de la interfaz **`Callable`**.
Para ver como implementar esta interfaz hacen falta algunos conceptos que todavía no hemos visto. En el tema de hilos de ejecución lo veremos con detalle.