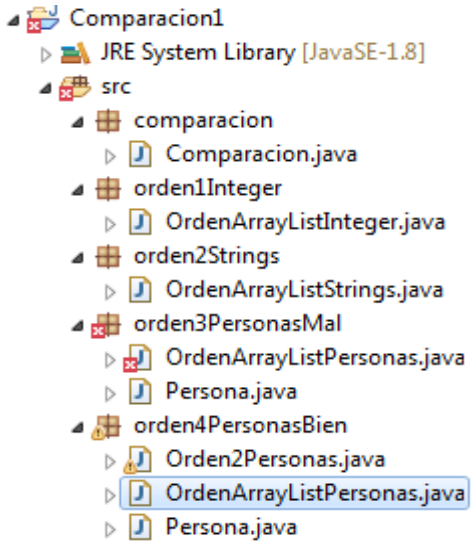
ORDENACIÓN UTILIZANDO LOS INTERFACES DE JAVA COMPARABLE Y COMPARATOR

Introducción

Vamos a ver algunos ejemplos para centrarnos en cuál es el problema que tenemos que resolver (**Proyecto Comparacion1**).



Recordemos cómo funciona el método compareTo, que hemos usado muchas veces (paquete **comparacion**):

```
public class Comparacion {

        public static void main(String[] args){
        Integer num = new Integer(6);

       Int resul = num.compareTo(new Integer(10));
        if (resul == 0)
                        System.out.println("Son iguales");
        else if (resul > 0)
                        System.out.println("num es mayor");
        else
                        System.out.println("num es menor");

       resul = "adios".compareTo("Hola");
        if (resul == 0)
                        System.out.println("Son iguales");
        else if (resul > 0)
                        System.out.println("adios es mayor");
        else
                       System.out.println("adios es menor");
        }

}
```

Salida:

num es menor
adios es mayor

Fíjate que la salida te dice si ponemos adiós que es mayor que Hola y si ponemos Adios nos saldría que es menor. Esto se debe a que en la tabla ASCII se colocan primero las mayúsculas y luego las minúsculas y ese el **orden natural** que usa compareTo.

Lo mismo que para los números, el 1 siempre será menor que el 2.

En el **ejemplo** del paquete **orden1Integer** vemos que la clase **Collections tiene un método estático llamado sort()**, al que pasándole por parámetro una colección, le decimos que la ordene. Si no especificamos el orden, el criterio es el mismo que el que utiliza el método compareTo() que ya conocemos (ya veremos cómo podemos especificar el orden).

En el **ejemplo** del paquete **orden2Strings** vemos cómo podemos ordenar también un ArrayList de Strings.

En estos dos ejemplos nos fijamos en que la forma de ordenar cuando queremos hacerlo al revés que con el orden natural es añadiendo al método sort un segundo parámetro:

```
Collections.sort(arrayListInt, Collections.reverseOrder());
```

Collections.reverseOrder() nos devuelve un objeto de tipo Comparator que nos permite ordenar la colección (o Array) de forma opuesta a la natural. Por lo que vemos que ahora el orden es descendente.

En caso de tener un Array haríamos lo siguiente para ordenar de las dos formas:

```
Arrays.sort(a);
Arrays.sort(a, Collections.reverseOrder());
```

En el **ejemplo** del paquete **orden3PersonasMal** vemos que tenemos un ArrayList de personas y que como tiene varios atributos en el objeto que queremos comparar, pues sort no sabe cómo tiene que ordenar (y no ordena, porque además da un error de compilación al intentar hacerlo).

Uso del interface Comparable y método compareTo (API Java) para comparar y ordenar objetos y colecciones

Ya vimos con los algoritmos de ordenación cómo podemos ordenar Arrays o ArrayList (en general cualquier colección).

Veremos que no es necesario incluir en nuestro código esos métodos, ya que en Java tenemos la posibilidad de ordenar de una manera más sencilla. Es el propio Java quien ordenará por nosotros, tal y como vimos en los dos ejemplos de la Introducción. Eso sí, tenemos que entender en qué se basa esa ordenación para ser capaces de utilizar la de manera adecuada.

En el ejemplo del paquete **orden3PersonasMal,** ya no nos funciona porque no hemos indicado qué criterio usamos para decidir la comparación (se trata de una clase que tiene varios atributos ahora).

Vamos a ver cómo una clase debe hacer uso de la interfaz **Comparable**. Esto nos va a permitir que en dicha clase se pueda realizar la comparación de objetos, permitiendo hacer **ordenaciones** de los mismos. Seguimos con la clase Persona de antes:

```
public class Persona {
    public int nombre, edad;
    public Persona( int nombre, int edad) {
       this.nombre = nombre;
       this.edad = edad;
   }

}
```

Lo primero que tenemos que pensar es que si queremos ordenar personas, tenemos que decidir cuál es el criterio para decidir que una persona es mayor, menor o igual a otra. En este caso podríamos decir que una persona es menor si tiene una edad menor, pero es una decisión que depende del caso en concreto. También podría ser válido si quisiéramos hacer un listado de personas ordenar por el nombre para hacer un listado alfabético.

Vemos ahora el paquete **orden4PersonasBien:**

En **OrdenDosPersonas.java** vemos que lo primero es implementar el interfaz Comparable:

```
public class Persona implements Comparable<Persona>
```

La base para realizar la ordenación va a ser la **interfaz Comparable**, en este primer ejemplo.

```
public interface Comparable<T>
```

Que tiene el método:

```
int compareTo(T o)
```

¿Qué significa la <T>? Representa que se puede aplicar a cualquier Tipo de Objeto, y que hay que especificarlo como hemos visto antes.

Vamos a implementar la interfaz Comparable en la clase que representa los objetos que queremos ordenar. Por ello, estamos obligados a implementar (sobrescribir) el método de la interfaz:

public int compareTo (Persona o)

Este método debe devolver un número negativo, cero o un número positivo en función de que el objeto que comparemos con el objeto "o" de referencia sea menor, igual o mayor respectivamente que ese objeto de referencia.

Por tanto, vamos a incluir el método **public int compareTo(Persona o)** en nuestra clase **Persona**.

```
@Override
 public int compareTo(Persona o) {
                    return this.nombre.compareTo(o.nombre);
  }
```

En el main de **OrdenDosPersonas.java** vemos:

```
Persona p1 = new Persona("Pepe", 28);
Persona p2 = new Persona("Juan", 32);
Persona p3 = new Persona("Juan", 32);

System.out.println(p1.compareTo(p2));
System.out.println(p2.compareTo(p1));
System.out.println(p2.compareTo(p3));
```

La salida será:

```
6
-6
0
```

Esa salida refleja, por ejemplo, que Pepe es mayor que Juan, ya que está después en el alfabeto. En concreto, lo que refleja es que hay 6 posiciones de diferencia en el alfabeto.

Nos devuelve esto porque hemos programado en nuestro método **compareTo** que compare así el atributo nombre de las personas.

En el mismo paquete, tenemos **OrdenArrayListPersonas.java,** en el que ahora lo que tenemos es una colección de Personas. El método estático **sort** de la clase **Collections** nos va a ordenar dicha colección en base al método compareTo que hemos programado. Ejecútalo y observa la salida.

1.- ¿Cómo hacemos si queremos la ordenación alfabéticamente en orden descendente?

2.- ¿Cómo hacemos si queremos la ordenación por edad en orden ascendente?

3.- ¿Cómo hacemos si queremos la ordenación por edad en orden descendente?

RESUMEN

Si tenemos 2 Personas y queremos compararlas u ordenarlas, la pregunta a hacerse sería:

"cuándo una persona es mayor que otra o cuándo es menor, o cuándo son iguales", atendiendo a algún tipo de atributo. Después sobreescribimos compareTo de acuerdo al criterio establecido.

Cuando en lugar de comparar explícitamente 2 personas lo que queremos es que se ordene una colección, lo que hay que hacer es sobrescribir el método compareTo, que es el que aplica el método sort para ser capaz de ordenar una colección o un array. Esto es lo que hemos visto en **OrdenArrayListPersonas.java,** cuando hace:

Collections.sort(personas);

En el proyecto **Comparacion2**, podemos ver un ejemplo (paquete ordenArrayPersonas) de cómo se puede hacer una ordenación en un array:

Arrays.sort(arrayPersonas);

En el proyecto **Comparacion3**, vemos otro ejemplo (paquete ordenArrayLPersonas) de cómo se puede hacer una ordenación en un ArrayList:

En este caso la clase Persona quedaría de la siguiente manera:

```
public class Persona implements Comparable<Persona>{
        public int dni, edad;
         public Persona( int d, int e){
         this.dni = d;
         this.edad = e;
}
```

Como podemos ver resaltado hemos incluido también el código **implements Comparable<Persona>**, que nos indica que vamos a implementar en la clase la interfaz Comparable para el parámetro Persona.

También hemos incluido el código necesario para implementar el método **public int compareTo(Persona o),** donde escribiremos el criterio para comparar personas. Si consideramos que el orden será **ascendente por dni**, tenemos que escribir:

```
@Override
public int compareTo(Persona o) {
        return (new Integer(this.dni).compareTo(new Integer(o.dni)));
}
```

O también:

```
@Override
public int compareTo(Persona o) {
        if (this.dni > o.dni)
                return 1;
        else if (this.dni < o.dni)
                return -1;
        else
                return 0;
}
```

Date cuenta de que **lo que se compara en compareTo deben ser objetos**.

Ahora escribimos un programa que generará 2 Personas y las compara mostrando cuál es la relación de comparación entre ellas (en base a los dnis).

```
public class Programa {
    public static void main(String arg[]) {
        Persona p1 = new Persona(74999999, 35);
        Persona p2 = new Persona(72759474, 30);

        if (p1.compareTo(p2) < 0 ) { System.out.println("La persona p1: es menor."); }
        else if (p1.compareTo(p2) > 0 ) {System.out.println("La persona p1: es mayor."); }
        else { System.out.println ("La persona p1 es igual a la persona p2"); }
    }
}
```

Comprueba con varios casos si funciona bien.

Si queremos ordenar un ArrayList, vemos que funcionaría perfectamente:

```
ArrayList<Persona> personas = new ArrayList<Persona>();

personas.add(new Persona(74999999, 35));
personas.add(new Persona(72759474, 30));
personas.add(new Persona(11111111, 35));
personas.add(new Persona(22222222, 30));
personas.add(new Persona(33333333, 35));
personas.add(new Persona(44444444, 30));
personas.add(new Persona(55555555, 35));
personas.add(new Persona(66666666, 30));

// Ahora sí nos deja ordenar personas
Collections.sort(personas);
```

Comprueba la salida.

Si queremos ordenar descendentemente por dni:

Collections.sort(personas, Collections.reverseOrder());

El reverseOrder() lo hace según lo que hayamos escrito en el método sobreescrito compareTo(),

Otra forma de ordenar descendentemente, sin usar reverseOrder() sería dar la vuelta a nuestra comparación:

```
@Override
public int compareTo(Persona o) {
        return (new Integer(o.dni).compareTo(new Integer(this.dni)));
        // hemos intercambiado el orden de comparación
}
```

Esta **implementación de la interfaz Comparable** es muy habitual ya que en general muchas veces vamos a desear que nuestras clases sean comparables no solo por el hecho de poder comparar objetos de dicha clase, sino por el uso de poder ordenar objetos de esa clase. La **ordenación** es una de las funciones más importantes y necesarias en el ámbito de la informática, ya que tener la información ordenada hace mucho más rápidas las consultas o modificaciones de datos.

¿Por qué crees que se usa la implementación de **interfaces** en Java?

Hay varios motivos, pero uno principal es que mediante la implementación de interfaces todos los programadores usan el mismo nombre de método y estructura formal para comparar objetos (o clonar, u otras operaciones).

Imagínate que estás trabajando en un equipo de programadores y tienes necesidad de utilizar una clase que ha codificado otro programador. Gracias a que se implementan interfaces Java, no necesitarás estudiar el código para ver qué método hay que invocar para comparar objetos de esa clase. Dado que todos los programadores usan la implementación de interfaces, sabemos que la comparación de objetos (o clonación, etc.) debe hacerse invocando determinados métodos de determinada forma. Esto facilita el desarrollo de programas y la comprensión de los mismos, especialmente cuando hablamos de programas o desarrollos donde pueden intervenir cientos o miles de clases diferentes.

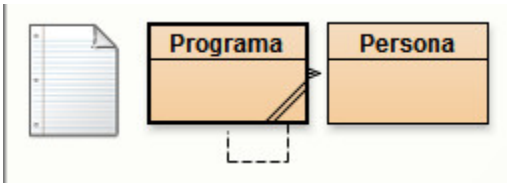
En el mismo caso de la clase Persona, vamos a cambiar el criterio de ordenación:

Ahora vamos a considerar que ordenamos a las personas por su edad, pero en el caso de que dos personas tengan la misma edad, vamos a considerar que es menor la que tenga un número de DNI más bajo. Para este caso debes introducir más personas que cumplan estas condiciones, con la misma edad.

```
@Override

public int compareTo(Persona o) {
        if (this.edad > o.edad)
                return 1;
        else if (this.edad < o.edad)
                return -1;
        else if (this.dni > o.dni)
                return 1;
        else if (this.dni< o.dni)
                return -1;
        else
                return 0;
}
```

Si cambiamos el criterio para que se ordene el ArrayList descendentemente por edad y ascendentemente por DNI ¿Cómo se haría?

El diagrama de clases de este ejemplo sería:



Interface Comparator (API java). Diferencias con Comparable. Clase Collections. Código ejemplo

INTERFACE COMPARATOR

A continuación procederemos a describir de manera detallada el uso y el comportamiento de la **interface Comparator** del api de Java.

Además veremos un ejemplo de uso de esta interfaz y cómo se utiliza. Señalaremos también cuáles son las **diferencias entre la interface Comparator y la interface Comparable**, dos interfaces que tienen ciertas similitudes y ciertas diferencias.

COMPARATOR

Esta interfaz es la encargada de permitirnos el poder comparar 2 elementos en una colección. Por tanto pudiera parecer que es igual a la interfaz **Comparable** (recordemos que esta interfaz nos obligaba a implementar el método **compareTo** (Object o)) que hemos visto anteriormente en el paquete java.lang.

Su diferencia es:

Mientras que **Comparable** nos obliga a implementar el método:

compareTo (Object o)

la interfaz **Comparator** nos obliga a implementar el método:

compare (Object o1, Object o2)

Si, por ejemplo para la clase Persona quisiéramos ordenar a las Personas por su altura, o por su nombre en orden alfabético podríamos recurrir a la interfaz **Comparator** para implementar el método **compare** (Object o1, Object o2) definiendo el método deseado.

EJERCICIO RESUELTO

Ahora vamos a desarrollar un ejercicio para comprender mejor su funcionamiento. Vamos a suponer una clase **Persona** similar a la vista en anteriores ocasiones sobre la cual vamos a definir un orden para poder ordenar colecciones de ese elemento.

Al igual que antes, usaremos la clase ArrayList para contener una colección de objetos.

Lo primero será definir mediante código nuestra clase Persona que será la siguiente:

```
/* Ejemplo Interface Comparable*/

public class Persona implements Comparable<Persona> {
   private int idPersona;
   private String nombre;
   private int altura;

   public Persona (int idPersona, String nombre, int altura) {
      this.idPersona = idPersona;
      this.nombre = nombre;
      this.altura = altura;}

   @Override
   public String toString() {
      return "Persona-> ID: "+idPersona+" Nombre: "+nombre+" Altura: "+altura;}

   @Override
   public int compareTo(Persona o) {
      return this.nombre.compareTo(o.nombre);}

   public int getIdPersona() {return idPersona;}
   public String getNombre() {return nombre;}
   public int getAltura() {return altura;}
}
```

En ella podemos ver que cada objeto de la clase Persona tendrá como campos un idPersona, nombre y altura. No hemos detallado los métodos set correspondientes a cada propiedad porque trataremos de centrarnos en el código que resulta de interés para este caso concreto.

Observamos también que queremos que esta clase sea ordenada naturalmente como ya hemos visto anteriormente al implementar Comparable. En este caso la ordenación se basa en el campo nombre de las Personas como queda reflejado en el método compareTo.

Las colecciones de Persona se ordenarán por este método siempre que se invoque una forma de ordenación que use la ordenación natural (orden natural es el que se indica en el método compareTo de la clase). Como ejemplo tenemos el siguiente código:

```
import java.util.ArrayList;
import java.util.Collections;
public class Programa {
   public static void main(String arg[]) {
         ArrayList<Persona> listaPersonas = new ArrayList<Persona>();
         listaPersonas.add(new Persona(1,"Maria", 185));
         listaPersonas.add(new Persona(2,"Carla", 190));
         listaPersonas.add(new Persona(3,"Yovana", 170));
```

```
        Collections.sort(listaPersonas);  // Ejemplo uso ordenación natural
        System.out.println("Personas Ordenadas por orden natural: "+listaPersonas);
    }
}
```

La lista de Personas ha sido ordenada usando la invocación Collections.sort pasando como parámetro la colección y el orden utilizado en esta invocación es el orden natural definido en su clase, es decir, por el nombre de cada Persona. En este caso: Carla, María y Yovana de acuerdo al orden alfabético C, M, Y.

La clase **Collections** que hemos utilizado es una clase similar a la clase **Arrays** del api de Java. Puede ser invocada simplemente importándola y nos aporta distintos métodos estáticos, siendo únicamente necesario pasarle determinados parámetros para obtener un resultado.

Ahora bien, imaginemos que queremos ordenar a las Personas por un orden distinto, podríamos elegir la altura. Una forma sería modificar el método compareTo, como hacíamos en el apartado anterior.

Y otra forma consiste (y a partir de aquí es lo nuevo) en hacer uso de la **interfaz Comparator** de la siguiente manera:

Creamos una clase diferente para **implementar la interfaz Comparator**, donde **sobreescribimos el método compare()** y en la clase Persona ya no implementaremos la interfaz Comparable.

```
/* Ejemplo Interface Comparator */

import java.util.Comparator;

public class OrdenarPersonaPorAltura implements Comparator<Persona> {
   @Override
   public int compare(Persona o1, Persona o2) {
         return o1.getAltura() - o2.getAltura();
          // Devuelve un entero positivo si la altura de o1 es mayor que la de o2
   }
}
```

Hemos definido el orden así, si o1 < o2 en este caso queremos que o1 vaya delante de o2 (decimos que es “más pequeño) por ordenarlo de menor altura a mayor altura. Entonces debemos devolver un número negativo, 0 si son iguales y un número positivo si o2 es de menor altura.

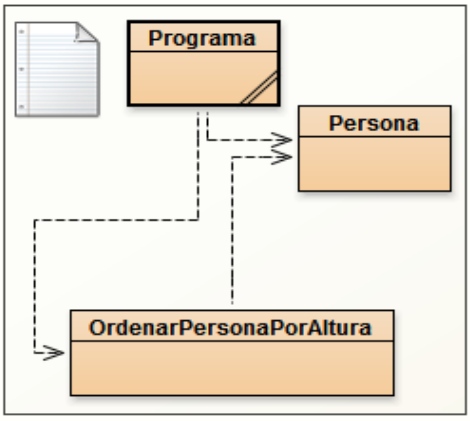
El siguiente paso es utilizar la clase OrdenarPersonaPorAltura de la siguiente manera en el programa principal:

```
public class Programa {
   public static void main(String arg[]) {
      ArrayList<Persona> listaPersonas = new ArrayList<>();

      listaPersonas.add(new Persona(1,"Maria",185));
      listaPersonas.add(new Persona(2,"Carla",190));
      listaPersonas.add(new Persona(3,"Yovana",170));

      Collections.sort(listaPersonas, new OrdenarPersonaPorAltura());

      System.out.println("Personas Ordenadas por orden total: " + listaPersonas);
     }
}
```

La única observación con respecto al anterior programa es que ahora la sentencia de ordenación Collections.sort lleva añadido el parámetro **new OrdenarPersonaPorAltura().**

Esta es otra forma del método sort de la clase Collections, en este caso pasándole como parámetros un objeto colección y un objeto que implemente Comparator y defina un orden a utilizar. Al pasarle un objeto de tipo OrdenarPersonaPorAltura indicamos el orden total a utilizar, que en este caso es la altura.

El diagrama de clases de este ejemplo es:



CONCLUSIÓN

Aunque **Comparable** y **Comparator** parecen iguales, en realidad no lo son, y mientras una se define para un orden por un solo criterio, la otra permite ordenar por varios criterios diferentes.

Gracias a la interfaz **Comparator**, podemos ordenar muy fácilmente colecciones utilizando clases que implementen el método compare **por cada tipo de ordenación que deseemos. Cosa que no podíamos hacer con la interfaz Comparable, ordenamos en el mismo programa solo de una forma, obviamente no es posible sobrescribir el método compareTo en la misma clase más de una vez.**

Prueba detenidamente los proyectos Jugadores1..Jugadores4 para terminar de comprender todo lo explicado.

En el apartado del Aula Virtual de **Excepciones, hicimos un ejercicio con Cuentas de bancos, en el que no desarrollasteis la pregunta:**

"Crea una **segunda versión** en la que las cuentas se puedan ver de dos formas distintas, ascendentemente por número de cuenta y descendentemente por saldo:"

```
1. Abrir cuenta
2. Ingresar dinero en cuenta
3. Sacar dinero de cuenta
4. Mostrar todas las cuentas(desc por saldo)
5. Mostrar todas las cuentas(asc por num. cuenta)
6. Mostrar una cuenta
7. Borrar una cuenta
8. Borrar todas las cuentas
9. Salir.


Elija opcion:
```

Completa el ejercicio con esas opciones.