APRENDERAPROGRAMAR.COM

INTERFACE COMPARATOR. DIFERENCIAS ENTRE COMPARATOR Y COMPARABLE. CLASE COLLECTIONS. EJERCICIOS RESUELTOS. (CU00915C)

Sección: Cursos
Categoría: Lenguaje de programación Java nivel avanzado I

Fecha revisión: 2029

Resumen: Entrega nº15 curso “Lenguaje de programación Java Nivel Avanzado I”.

Autor: Manuel Sierra

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 estupendo ejemplo de uso de esta interfaz y cómo se utiliza a modo profesional. 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. Aunque es cierto que es similar, estas dos interfaces en absoluto son iguales, sino ¿qué sentido tendría su existencia?

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).

El primer método nos sirvió para implementar un método de comparación en una clase de ejemplo a la que denominamos Persona. La implementación de este método se hace para indicar el orden natural de los elementos de esa clase. Pero ¿qué ocurre si queremos ordenar los elementos por otro orden?

Por ejemplo para la clase Persona lo lógico es tomar como orden natural por ejemplo la edad de esa Persona. Pero ¿qué ocurre si queremos ordenar las Personas digamos por su altura, o por su nombre en orden alfabético? Pues que el orden natural definido no nos sirve y debemos de 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 en la anterior entrega, 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 Comparator aprenderaprogramar.com */

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);}
}
```

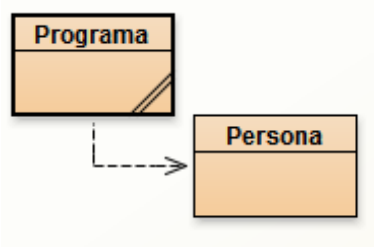
En ella podemos ver que cada objeto de la clase Persona tendrá como campos un idPersona, nombre y altura. No hemos detallado los métodos get y set correspondientes a cada propiedad porque trataremos de centrarnos en el código que resulta de interés para este caso concreto.

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. Es decir, cuando se trate de realizar una ordenación “natural”, esta se hará por orden alfabético de nombres.

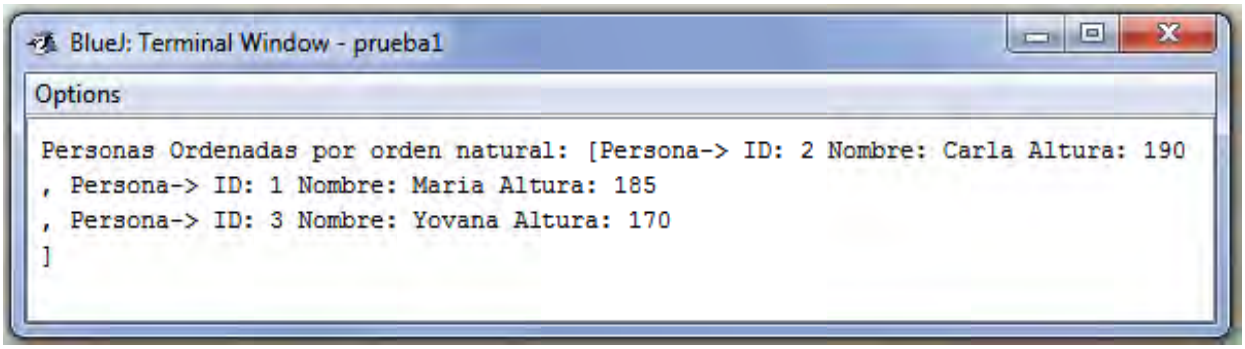
Las colecciones de Persona se ordenarán por este método siempre que se invoque una forma de ordenación que use la ordenación natural. Como ejemplo tenemos el siguiente código:

```
/* Ejemplo Interface Comparator aprenderaprogramar.com */

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);
     }
}
```

El diagrama de clases de este ejemplo previo es:



Y el resultado tras ejecutar el programa es:



Donde vemos cómo efectivamente 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 a lo que hemos denominado orden natural, en este caso hablaremos de orden total. Como orden total podríamos elegir su fecha de nacimiento o en este caso por ejemplo usaremos la altura.

Entonces tenemos que hacer uso de la interfaz Comparator de la siguiente manera:

```
/* Ejemplo Interface Comparator aprenderaprogramar.com */
import java.util.Comparator;

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

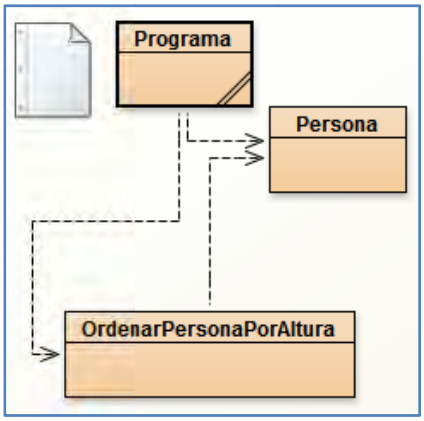
Observar que hemos definido el orden así, si o1 < o2 en este caso queremos que o1 vaya delante de o2 (decimos que es "más pequeño) por ordenarlo de menor altura a mayor altura. Entonces debemos devolver un número negativo, 0 si son iguales y un número positivo si o2 es de menor altura.

Ahora bien con esto solo no es suficiente, ya que hay que utilizar la clase OrdenarPersonaPorAltura de la siguiente manera en el programa principal:

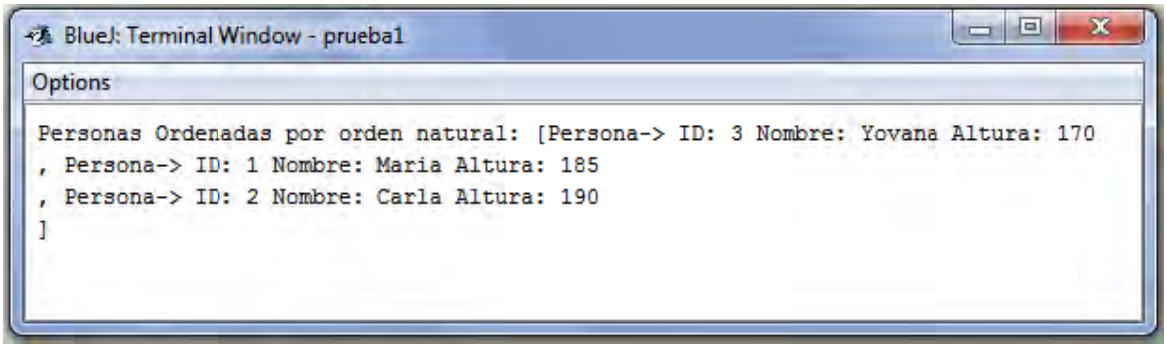
```
/* Ejemplo Interface Comparator aprenderaprogramar.com */

import java.util.ArrayList;
import java.util.Collections;

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 natural: "+listaPersonas);
    }
}
```

La única observación con respecto al anterior programa es que ahora la línea 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 total 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:



Y el resultado tras ejecutar el programa es:



Comprobamos que el resultado de ordenación no es igual cuando hemos utilizado la interface Comparable y el método compareTo que cuando hemos utilizado la interface Comparator y el método compare.

CONCLUSIONES

Una conclusión puede ser que aunque parecen iguales **Comparable** y **Comparator**, en realidad no lo son y mientras una se define para el orden natural, la otra se define para un orden total respectivamente. El orden natural es utilizado por diversos métodos del api de Java, pero no siempre hemos de usar el orden natural. En determinados casos podremos querer ordenar objetos por un orden distinto al orden natural, y para ello nos será útil implementar la interface Comparator.

La segunda conclusión es que gracias a la interfaz **Comparator**, podemos ordenar muy fácilmente colecciones utilizando clases que implementen el método compare por cada tipo de ordenación que deseemos.

Próxima entrega: CU00916C

Acceso al curso completo en aprenderaprogramar.com -- > Cursos, o en la dirección siguiente: http://aprenderaprogramar.com/index.php?option=com_content&view=category&id=58&Itemid=180