COMPARATOR

Ejercicio U7_B6A_E1:
Modifico la lógica de compare() de forma que ahora devuelva un negativo si p1.edad>p2.edad para colocar primero la persona de edad mayor.

```
class ComparadorDePersonas implements Comparator<Persona> {
   @Override
   //ordeno las personas por edad ascendente
   public int compare(Persona p1, Persona p2) {
      if (p1.edad>p2.edad){
         return -1;
      }else if (p1.edad<p2.edad){
         return 1;
      }else{
         return 0;
      }
   }
}
```

El siguiente código es más conciso y hace lo mismo pero es menos legible y "quizá" no merece la pena

```
class ComparadorDePersonas implements Comparator<Persona> {
   @Override
   //ordeno las personas por edad ascendente
   public int compare(Persona p1, Persona p2) {
         return p2.edad-p1.edad;
   }
}
```

Ejercicio U7_B6A_E2: Modifica el ejemplo anterior para que imprima las personas por orden ascendente de nombre.

```
class ComparadorDePersonas implements Comparator<Persona> {
   @Override

   //ordeno las personas por nombre ascendente
   public int compare(Persona p1, Persona p2) {

    if (p1.nombre.compareTo(p2.nombre) <0)
       return -1;
    else if (p1.nombre.compareTo(p2.nombre) >0)
       return 1;
    else return 0;
   }
}
```

Como el método de Strings compareTo() tiene la misma lógica que el comportamiento Standard de compare(), es decir si obj1<obj2 devuelve negativo, si obj1>obj2 positivo, si son iguales cero podemos simplificar el código

```
public int compare(Persona p1, Persona p2) {
      return (p1.nombre.compareTo(p2.nombre));
 }
```

Ejercicio U7_B6A_E3: Modifica el ejemplo anterior para que imprima las personas por orden descendente de nombre.

```
class ComparadorDePersonas implements Comparator<Persona> {
   @Override

   //ordeno las personas por nombre ascendente
   public int compare(Persona p1, Persona p2) {

    if (p1.nombre.compareTo(p2.nombre) > 0)
        return -1;
    else if (p1.nombre.compareTo(p2.nombre) < 0)
        return 1;
    else return 0;
   }
}
```

O más simple pero creo que poco claro:

```
public int compare(Persona p1, Persona p2) {
      return - (p1.nombre.compareTo(p2.nombre));
 }
```

Ejercicio U7_B6A_E4:
El comparador siempre devuelve 0 y por tanto siempre dice que los objetos son iguales. Comienza insertando el primer objeto que es "Elías", pero los siguientes como el comparador les dice que son iguales no los inserta. Recuerda que los Set no insertan duplicados.

Ejercicio U7_B6A_E5:¿Puedo sustituir en los ejercicios anteriores TreeSet por HashSet?
No, si consultas el API puedes observar que a ningún constructor de HashSet se le puede pasar un comparador. HashSet almacena y por tanto indirectamente ordena, en base a funciones hash internas

Ejercicio U7_B6A_E6:

En principio, en los *Set* no tiene porqué ser importante obtener un listado de ellos en orden, ya que matemáticamente, un conjunto no tiene orden, y se supone, que si utilizas un conjunto no vas a requerir orden. Observa que TreeSet es un conjunto pero podemos obtener sus elementos ordenados aprovechándose de cómo se almacenan internamente , pero el objetivo de este almacenamiento es acceder a elementos individuales muy rápidamente cuando busco un elemento por un valor, no específicamente obtener una lista ordenada.

Por tanto HashSet no es una clase apropiada para obtener los elementos en orden. Para obtener una lista ordenada de elementos hay otras colecciones, como el TreeSet. Si por la razón que fuera, en nuestro contexto HashSet es la clase ideal, pero al mismo tiempo necesito en un momento dado ver sus elementos ordenados tengo que programar esta impresión ordenada. Por ejemplo, pasar el HashSet a un TreeSet para hacer la impresión.

```
import java.util.HashSet;
import java.util.TreeSet;
```

```
public class App{
   public static void main(String[] args){
      HashSet<Integer> hs = new HashSet<>() ;
      hs.add(1) ;
      hs.add(22) ;
      hs.add(12) ;
      System.out.println("hs: "+hs);
      TreeSet<Integer> ts= new TreeSet<>(hs);
      System.out.println("ts: "+ts);

   }
}
```