SOBRE LA NECESIDAD DE COMPARATOR/COMPARABLE PARA ORDENAR

Como una colección es una secuencia de elementos a menudo a esos elementos queremos aplicarles el concepto de "orden".

Por ejemplo, para ordenar un array de números enteros lo único que tenemos que hacer es invocar el método static sort() de la clase Arrays.

Ejemplo:

```
import java.util.Arrays;
class App{
  public static void main(String args[]){
    double[] notas = {8.5,7.0,6.0,9.2};
    Arrays.sort(notas);
    for(double d:notas)
        System.out.print(d+" ");
  }
}
```

Si el array es de objetos, por ejemplo de objetos Double también O.K.

```
import java.util.Arrays;
class App{
  public static void main(String args[]){
    Double[] notas = {8.5,7.0,6.0,9.2};
    Arrays.sort(notas);
    for(Double d:notas)
        System.out.println(d+" ");
  }
}
```

Si es de objetos String ¡también O.K.!

```
import java.util.Arrays;
class App{
  public static void main(String args[]){
    String[] notas = {"UNO","DOS","TRES"};
    Arrays.sort(notas);
    for(String s:notas)
        System.out.println(s+" ");
  }
}
```

Pero imaginemos ahora que tenemos que ordenar un array de objetos Personas donde una persona se describe con los atributos nombre y edad. ¿Cómo ordenamos a las personas, por Nombre o por edad? ¡Depende de lo que queramos!. Si probamos

```
import java.util.Arrays;
class Persona{
    String nombre;
    int edad;

    public Persona(String nombre, int edad) {
        this.nombre = nombre;
        this.edad = edad;
    }
 }
class App{
  public static void main(String args[]){
    Persona[] personas = {new Persona("yo",44), new Persona("tu",37)};
    Arrays.sort(personas);
```

```
    System.out.println(personas[0].nombre+" "+personas[0].edad);
  }
}
```

Obtendremos un error de ejecución con referencia al número de línea de

Arrays.sort(personas);

¡el método sort() no sabe ordenar personas.!

Otro ejemplo, TreeSet, sabemos que almacena los elementos ordenados, pero, ¿Con qué orden? Pues digamos que con el orden "natural:*1* es menor que *2*, *"a"* es menor que *"b"*, …

```
import java.util.TreeSet;
class App {
    public static void main(String[] args) {
       TreeSet<Integer> tp= new TreeSet<>();
       tp.add(20); tp.add(1); tp.add(13);
       System.out.println(tp);
    }
}
```

Realmente no es que TreeSet sepa ordenar Objetos Integer, lo que realmente ocurre es que la clase Integer le dice a TreeSet como hacerlo y lo hace a través del interface *Comparable.* Observa en el API como Integer implementa Comparable

Class Integer

java.lang.Object
  java.lang.Number
    java.lang.Integer

All Implemented Interfaces:

Serializable, Comparable<Integer>

Entenderemos mejor esto en el próximo boletín ya que en este boletín el interface que vamos a usar es *Comparator.*

¿Y si queremos un TreeSet de Personas, ordenadas por Edad?. También salta una excepción, ya que TreeSet inserta ordenado y desconoce cual es el orden de las personas

```
import java.util.TreeSet;

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

class App {
    public static void main(String[] args) {
       TreeSet<Persona> tp= new TreeSet<>();
       tp.add(new Persona("yo",99));
    }
}
```

Como indicar que un objeto obj1 "es menor, igual o mayor" que otro objeto obj2
para poder indicar la relación de orden entre dos objetos tenemos dos mecanismos:

- Interface Comparator:
- Interface Comparable

Quien ordena
Depende, puede ser un método static cómo sort(), el código interno de una colección como TreeSet, etc. pero haga quien lo haga necesita un criterio de comparación de dos objetos que es lo que se escribe con Comparator/Comparable.

COMPARATOR
Consulta el api!!!
Observa cosas importantes:

- Comparable es genérico
- Para comparar dos objetos ambos deben ser del mismo tipo T
- El código de comparación se escribe en el método *int compare(T obj1, T obj2)*

El comportamiento *"esperado standard"* de este método debe ser:

- Devuelve 0 si los objetos son iguales
- Devuelve un valor positivo si obj1 es mayor que obj2
- Devuelve un valor negativo, si obj1 es menor que obj2.

Si escribimos un compare() y que no cumple este protocolo volveremos locos a los programadores que usen nuestro código pues ellos cuentan que el comportamiento de todo compare() como el que se indica la API java.

Ejemplo de uso de Comparator con TreeSet
Al trabajar con un TreeSet de objetos Persona, me encuentro con problemas en tiempo de ejecución:

```
import java.util.TreeSet;

class Persona{
    String nombre;
    int edad;

    Persona(String nombre, int edad) {
        this.nombre = nombre;
        this.edad = edad;
    }
}
public class App {
    public static void main(String[] args) {
        TreeSet<Persona> tp = new TreeSet<>();
        tp.add(new Persona("Elías",5));
        tp.add(new Persona("Román",4));
        tp.add(new Persona("Telma",1));
        System.out.println("TreeSet de personas: "+ tp);
    }
}
```

Al ejecutar:

```
run:
```

```
Exception in thread "main" java.lang.ClassCastException: Persona cannot be cast to java.lang.Comparable
        at java.util.TreeMap.compare(TreeMap.java:1188)
        at java.util.TreeMap.put(TreeMap.java:531)
        at java.util.TreeSet.add(TreeSet.java:255)
        at App.main(App.java:15)
Java Result: 1
```

La línea 15 de mi main() se corresponde con
tp.add(new Persona("Elías",5));

SOLUCIÓN:
Hay dos soluciones básicas:

- Hacer que la clase Persona implemente Comparable. Esta solución la vemos en el siguiente boletín.
- Crear una clase que implemente Comparator, crear un objeto de esta clase y pasar dicho objeto al constructor de TreeSet. Es lo que haremos a continuación.

Observa la siguiente versión de constructor de TreeSet

```
public TreeSet(Comparator<? super E> comparator)

Constructs a new, empty tree set, sorted according to the specified comparator. .
```

Vamos a utilizar esta versión de constructor para lo que previamente tenemos que crear un objeto comparador.

```
import java.util.Comparator;
import java.util.TreeSet;

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

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;
        }
    }
}

public class App {
    public static void main(String[] args) {
        TreeSet<Persona> tp = new TreeSet<>(new ComparadorDePersonas());
        tp.add(new Persona("Elías",7));
        tp.add(new Persona("Román",6));
        tp.add(new Persona("Telma",3));
        String sImprimir="TreeSet de personas por edad ascendente \n";
        for(Persona p:tp)
            sImprimir=sImprimir+p.nombre+ ", "+p.edad+"\n";
        System.out.println(sImprimir);
```

```
  }
}
```

```
run:
TreeSet de personas por edad ascendente
Telma, 3
Román, 6
Elías, 7
```

RESUMEN DEL CÓDIGO ANTERIOR AL RESPECTO DE DEFINIR ORDEN DE ELEMENTOS EN EL TREESET

1. Se crea una clase que implemente la interface Comparator para redefinir el método compare()

   class ComparadorDePersonas implements Comparator<Persona> {
   etc...

2. En compare() definimos el orden a nuestro gusto, sabiendo que si devuelve un número negativo los métodos de almacenamiento de TreeSet colocan el primer objeto antes que el segundo.

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

3. Al crear el TreeSet, en el constructor se pasa por parámetro un objeto de la clase comparadora creada anteriormente

TreeSet<Persona> tp = new TreeSet<Persona>(new ComparadorDePersonas());

Tienes que tener claro que TreeSet también conoce y sigue "el standard" que marca la interface Comparator y por eso cuando nosotros le pasamos en el constructor un objeto Comparator "sabe" que debe invocar al método compare() para decidir el orden

Por lo tanto, el programador de App(YO) y el programador de TreeSet(un empleado de oracle) nos comunicamos a través del interface comparator. Ambos seguimos el protocolo: yo escribo compare(), y el otro utiliza compare(). Yo lo escribo como me indica el API(para decir que obj1 va antes que obj2 devuelvo un negativo etc.) y el otro programador cuenta con que hice así

Ejercicio U7_B6A_E1: Modifica el ejemplo anterior con TreeSet para que imprima las personas por orden descendente de edad.

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

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

Ejercicio U7_B6A_E4: Modifica el ejemplo anterior de forma que use el siguiente comparador y explica porqué el resultado es un TreeSet que sólo consta de la persona "Elías"

```
class ComparadorDePersonas implements Comparator<Persona> {
    @Override
    public int compare(Persona o1, Persona o2) {
        return 0;
    }

}
```

Ejercicio U7_B6A_E5:¿Puedo sustituir en los ejercicios anteriores TreeSet por HashSet?

Ejercicio U7_B6A_E6: Imprimir un HashSet ordenado.

Crea un HashSet de enteros. Ya que un TreeSet en su constructor admite como parámetro una colección, por ejemplo un HashSet, crea un TreeSet para imprimir el HashSet ordenado

Ejemplo de uso de Comparator para ordenar arrays con Arrays.sort()
El método genérico sort() tiene una versión para especificar comparador

```
public static <T> void sort(T[] a,
            Comparator<? super T> c)
```

Sorts the specified array of objects according to the order induced by the specified comparator.

```
import java.util.Arrays;
import java.util.Comparator;


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

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;
        }
    }
}

class App {
    public static void main(String[] args) {
        Persona[] personas = {new Persona("yo",44), new Persona("tu",37)};
        Arrays.sort(personas, new ComparadorDePersonas());
        System.out.println(personas[0].nombre+" "+personas[0].edad);

    }
}
```

Ejemplo de uso de Comparator para ordenar listas con Collections.sort()

La versión que usa comparador(consulta API)
Collections.sort(objeto a ordenar, comparador para ordenar el objeto anterior)

```
public static <T> void sort(List<T> list,  Comparator<? super T> c)
```

Crear una LinkedList de artículos e imprimirla ordenada utilizando Collections.sort() y Comparator.

```
import java.util.Collections;
import java.util.Comparator;
import java.util.LinkedList;

class Articulo {
    String codArticulo;
    String descripcion;
    int cantidad;
    Articulo(String codArticulo, String descripcion, int cantidad) {
        this.codArticulo = codArticulo;
        this.descripcion = descripcion;
        this.cantidad = cantidad;
    }


}

class ComparadorArticulos implements Comparator<Articulo>{
    @Override
    public int compare( Articulo o1, Articulo o2) { return o1.codArticulo.compareTo(o2.codArticulo); }
}

class App {

    public static void main(String[] args) {
        LinkedList<Articulo> articulos = new LinkedList<Articulo>();
        articulos.add(new Articulo("34","cuchillo",5));
        articulos.add(new Articulo("12","tenedor",7));
        articulos.add(new Articulo("41","cuchara",4));
        articulos.add(new Articulo("11","plato",6));

        Collections.sort(articulos, new ComparadorArticulos());
        for(Articulo a:articulos)
            System.out.println(a.codArticulo+", "+a.descripcion+", "+a.cantidad);
    }
}
```

Ejemplo de uso de Comparator para ordenar listas EN ORDEN INVERSO con Collections.sort()

En lugar de generar un nuevo comparador simplemente podemos invertir la lógica del comparador con el método Collections.reverseOrder(). Este método necesita en este caso como parámetro un comparador y devuelve un nuevo comparador con la lógica invertida

```
import java.util.Collections;
import java.util.Comparator;
import java.util.LinkedList;

class Articulo {
    String codArticulo;
    String descripcion;
    int cantidad;
    Articulo(String codArticulo, String descripcion, int cantidad) {
        this.codArticulo = codArticulo;
        this.descripcion = descripcion;
        this.cantidad = cantidad;
    }


}

class ComparadorArticulos implements Comparator<Articulo>{
    @Override
    public int compare( Articulo o1, Articulo o2) { return o1.codArticulo.compareTo(o2.codArticulo); }
}

class App {

    public static void main(String[] args) {
        LinkedList<Articulo> articulos = new LinkedList<Articulo>();
        articulos.add(new Articulo("34","cuchillo",5));
        articulos.add(new Articulo("12","tenedor",7));
        articulos.add(new Articulo("41","cuchara",4));
        articulos.add(new Articulo("11","plato",6));

        Collections.sort(articulos, Collections.reverseOrder(new ComparadorArticulos()));
        for(Articulo a:articulos)
            System.out.println(a.codArticulo+", "+a.descripcion+", "+a.cantidad);
    }
}
```