COMPARABLE

¡Consulta Comparable en el API!.
El objetivo es el mismo que el de comparator, pero ahora el código que indica cómo comparar dos objetos se escribe en en el método compareTo() y la interfaz la implementa la clase de los propios objetos que queremos ordenar.

Ejemplo de TreeSet con comparable

Observa:

- Ahora el código para comparar va en la propia clase Persona
- El constructor empleado para TreeSet es la versión sin parámetros. Esta versión sólo funciona si los objetos que agrupa el TreeSet implementan Comparable

```
import java.util.TreeSet;

class Persona implements Comparable<Persona>{
    String nombre;
    int edad;
    Persona(String nombre,int edad){
        this.nombre=nombre;
        this.edad=edad;
    }

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

    }

    @Override
    public String toString() {
        return "Persona [edad=" + edad + ", nombre=" + nombre + "]";
    }

}

class App {
    public static void main(String[] args) {
       TreeSet<Persona> tp= new TreeSet<>();
       tp.add(new Persona("yo",99));
       tp.add(new Persona("tu",50));
       System.out.println(tp);
    }
}
```

Ejemplo: ¿Por qué el siguiente código no da error y produce la salida ordenada ascendentemente?

```
import java.util.TreeSet;
class App {
    public static void main(String[] args) {
        TreeSet<Integer> t= new TreeSet<>();
        t.add(3);t.add(7);t.add(1);
        System.out.println(t);
     }
}
```

Consulta el API de Integer y comprueba que:

- implementa Comparable

> **All Implemented Interfaces:**
>
> `Serializable, Comparable<Integer>`

- Por lo tanto, tiene un método compareTo()
- compareTo() tiene lógica para orden ascendente de enteros

> **Returns:**
>
> the value 0 if this Integer is equal to the argument Integer; a value less than 0 if this Integer is numerically less than the argument Integer; and a value greater than 0 if this Integer is numerically greater than the argument Integer (signed comparison).

Por lo tanto, ¡¡ TreeSet utiliza el compareTo() de Integer para hacer la ordenación!! Y de ahí el resultado ordenado ascendentemente

Ejemplo de Arrays con comparable

Consideraciones similares a TreeSet

```
import java.util.Arrays;

class Persona implements Comparable<Persona>{
    String nombre;
    int edad;
    Persona(String nombre,int edad){
        this.nombre=nombre;
        this.edad=edad;
    }

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

    }

    @Override
    public String toString() {
        return "Persona [edad=" + edad + ", nombre=" + nombre + "]";
    }

}

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


    }
}
```

Ejemplo: ¿Por qué el siguiente código no da error y produce la salida *dos tres uno*?

```
import java.util.Arrays;
class App {
    public static void main(String[] args) {
        String[] misStrings = {"uno","dos","tres"};
        Arrays.sort(misStrings);
        for(String s:misStrings){
            System.out.print(s+" ");
        }
     }
}
```

Consulta el API de String y haz lo mismo que con el ejemplo de Integer, comprobando que String implementa Comparable, por lo tanto, tiene un método compareTo() y que la lógica de compareTo() nos dice si "alfabéticamente" this es menor que anotherString. Sort() al no recibir ningún Comparador por parámetro utiliza el compareTo() de String

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

Al método le pasamos un objeto que dentro tiene el código de compareTo()

Collections.sort(objeto a ordenar que implementa la interfaz Comparable)

Observa el api del método

```
public static <T extends Comparable<? super T>> void sort(List<T> list)
```

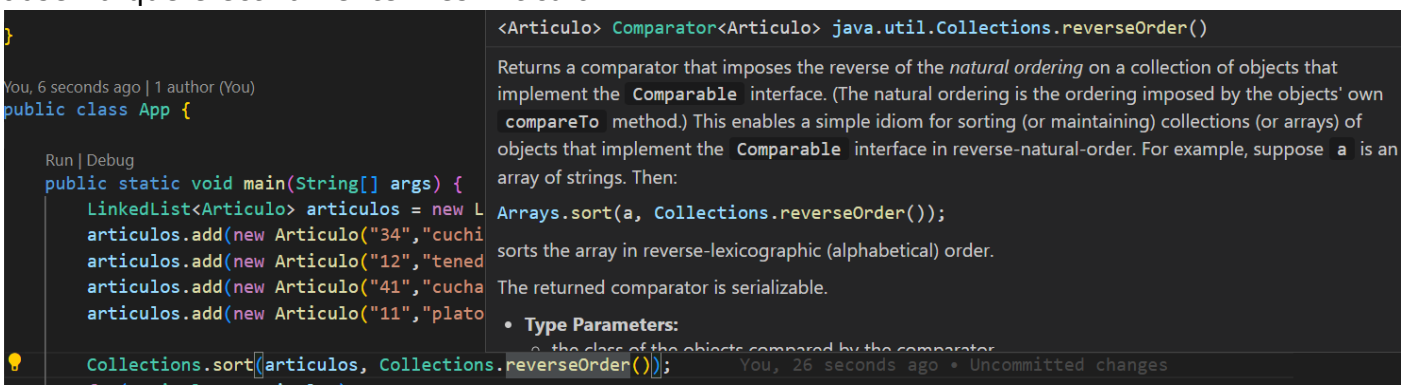
Se índica T extends Comparable. Eso quiere decir que el objeto de tipo T entre otras cosas implementa Comparable

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

class Articulo implements Comparable<Articulo> {
    public String codArticulo; // Código de artículo
    public String descripcion; // Descripción del artículo.
    public int cantidad; // Cantidad a proveer del artículo.
    //añado constructor por comodidad para App

    public Articulo(String codArticulo, String descripcion, int cantidad) {
        this.codArticulo = codArticulo;
        this.descripcion = descripcion;
        this.cantidad = cantidad;
    }
    @Override
   public int compareTo(Articulo o) { return codArticulo.compareTo(o.codArticulo); }

}

public class App {

    public static void main(String[] args) {
        LinkedList<Articulo> articulos = new LinkedList<>();
        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);
        for(Articulo a:articulos)
            System.out.println(a.codArticulo+", "+a.descripcion+", "+a.cantidad);
    }
```

```
}
```

¿sort() hace magia?
No!. Puedes observa en el API que sort() va a trabajar con un tipo T que implementa comparable y por tanto podrá invocar el código de compareTo(). Dado el contexto anterior, deduce que T es el tipo Articulo.

Ejemplo de uso de Comparable para ordenar listas EN ORDEN INVERSO con Collections.sort()
En lugar de cambiar el código de comparable, simplemente podemos invertir su lógica a través de un comparador que devuelve Collections.reverseOrder(). Si no indicamos al método ningún parámetro intenta deducir del contexto un Tipo comparable que contiene la lógica de ordenación y genera un comparador con la lógica contraria
En nuestro ejemplo deduce que el tipo comparable es artículo observa como en el IDE se observa que efectivamente T es Articulo



CONCLUSIÓN: El interfaz Comparator y Comparable los podemos ver como protocolos que ponen de acuerdo a dos o más programadores para ordenar objetos. Pensando en programadores, en los ejemplos anteriores:

1. El programador de la clase Articulo bien implementando Comparable o bien escribiendo una clase Comparator describe cuando un artículo va antes que otro, es decir, no ordena nada sino que simplemente indica el criterio de ordenación.
2. Los programadores del JDK con el criterio anterior, mueven los elementos de la colección para que todos cumplan el criterio anterior, es decir, los ordena aplicando eficientes algoritmos de ordenación

Ejercicio U7_B6B_E1: Sobre el ejemplo que usa
Collections.sort(articulos);

reescribe compareTo() para que se ordenen los artículos por cantidad ascendentemente(de menos cantidad a más cantidad).

Ejercicio U7_B6B_E2: Deseo ordenar una lista de teléfonos de la siguiente forma; Primero aparecerán ordenados de mayor a menor los números internacionales (con símbolo "+" delante) y después los locales, también ordenados de mayor a menor.

La clase String implementa el interfaz Comparable y por tanto el siguiente código es posible

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

```
public class App {
    public static void main(String[] args) {
        ArrayList<String> telefonos = new ArrayList<>();
        telefonos.add("981555555");
        telefonos.add("+34981565656");
        telefonos.add("666666666");
        telefonos.add("+34666666666");

        Collections.sort(telefonos);
        for(String tlf: telefonos)
            System.out.print(tlf+ "  ");
    }
}
```

Pero produce la salida:

```
run:
+34666666666  +34981565656  666666666  981555555  BUILD SUCCESSFUL (total time: 0 seconds)
```

El código anterior Ordena bajo el criterio de la clase String, pero no cumple el enunciado. Una solución para alcanzar el enunciado es utilizar un comparador.

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

public class App {
    public static void main(String[] args) {
        ArrayList<String> telefonos = new ArrayList<>();
        telefonos.add("981555555");
        telefonos.add("+34981565656");
        telefonos.add("666666666");
        telefonos.add("+34666666666");

        Collections.sort(telefonos, new ComparadorTelefonos());
        for(String tlf: telefonos)
            System.out.print(tlf+ "  ");
    }
}
```

```
run:
+34981565656  +34666666666  981555555  666666666  BUILD SUCCESSFUL (total time: 0 seconds)
```

SE PIDE: Escribe el comparador. Hay que tener en cuenta dos cosas:

- El + es tiene código unicode menor que los dígitos, por lo tanto, para listar primero números con + y ser de mayor o menor no basta con la comparación directa de Strings
- ¿Cómo interpretamos que string numérico es mayor que otro?.
  Recuerda que la comparación de String, independientemente de que los caracteres sean dígitos o no digitos se hace por "orden alfabético estilo diccionario" , o mejor dicho, "por orden de código de caracteres". De tal forma que por ejemplo:
  Pensando en enteros 111 > 12 es true, pero
  Pensando en Strings "111" > "12" es false ya que la "palabra" "111" iría antes en un diccionario que "12".

Por tanto, si queremos que el teléfono "12" sea mayor que "111" tengo que que escribir el comparador de forma que "111" > "12" sea false para lo que no puedo utilizar directamente la comparación de Strings.

COMPARATOR VS COMPARABLE

Para clases muy utilizadas que tienen un orden que claramente se va a usar muy frecuentemente lo ideal es que implementen comparable. Por ejemplo la clase String y la clase Integer implementan Comparable y es muy cómodo

Para otras clases como la clase Articulo del ejemplo anterior, el orden que se requiere de ellas es cambiante y ninguno predomina, entonces mejor no especificar ningún orden con comparable y los programadores que la usen especificarán el orden que quieren con Comparator

COLAS DE PRIORIDAD

Hay diferentes tipos de colas piensa por ejemplo en:

- Una cola de cine: es una cola clásica primero en llegar, primero en entrar. Bueno, supongamos que la peña no se cuela …
- Una cola de urgencias en el CHUS: es una cola de prioridad, se atiende por gravedad no por orden de llegada

La clase PriorityQueue en java implementa el concepto de "cola de prioridad". Su característica más importante es que sus elementos serán de un tipo comparable para tener un criterio de orden. El siguiente ejemplo es posible ya que String es Comparable. Recuerda que recorrer colas al quitar con poll() realmente debemos hablar de un vaciado

```
import java.util.PriorityQueue;

public class App {

    public static void main(String args[]) {
        PriorityQueue<String> cp= new PriorityQueue<>();

        cp.add("m"); cp.add("z"); cp.add("a");

        System.out.print(cp.poll()+" ");System.out.print(cp.poll()+" ");System.out.print(cp.poll()+" ");

        //otras formas de recorrer la cola
        cp= new PriorityQueue<>();
        cp.add("m"); cp.add("z"); cp.add("a");

        //System.out.println(cp); el toString() no me da el resultado esperado

        /* este for da resultados incongruentes por el poll va modificando el size
           estudiaremos estos casos más detenidamente cuando veamos iteradores
          for(int i=0;i<cp.size();i++){
            System.out.print(cp.poll()+" ");
        }*/
        System.out.println("");
        String s=cp.poll();
        while(s!=null){
            System.out.print(s+" ");
            s=cp.poll();
        }


    }

}
```

Ejercicio U7_B6B_E3: Escribe la clase Enfermo para que funcione el siguiente main

```
import java.util.PriorityQueue;
```

```
public class App {

    public static void main(String args[]) {
        PriorityQueue<Enfermo> enfermos= new PriorityQueue<>();
        enfermos.add(new Enfermo("Cristina",2));
        enfermos.add(new Enfermo("Juan",1));
        enfermos.add(new Enfermo("Ana",3));
         enfermos.add(new Enfermo("Oscar",3));
        System.out.println("");
        Enfermo e=enfermos.poll();
        while(e!=null){
            System.out.println(e);
             e=enfermos.poll();
        }


    }

}
```

SALIDA

```
Enfermo{nombre=Oscar, gravedad=3}
Enfermo{nombre=Ana, gravedad=3}
Enfermo{nombre=Cristina, gravedad=2}
Enfermo{nombre=Juan, gravedad=1}
```

La clase enfermo tendrá dos campos

String nombre;
int gravedad;//de 1 a 3. 3 más grave que 2. 2 más que 1.