

Ordenar en Streams

Hay muchas posibilidades, vemos sólo las más habituales.

El método `sorted()` de Stream

Es similar al método `sort()` de `Collections` que ya conocemos pero aplicado a los elementos de un Stream en lugar de a los elementos de un colección. Observa que para streams se llama **`sorted()`**. Igual que con `sort()` de `Collections`, habrá una versión sin comparador aplicable cuando los elementos del Stream sean comparables y otra con comparador

```
Stream<T> sorted()
```

```
Stream<T> sorted(Comparator<? super T> comparator)
```

Si la clase es `Comparable`, `sorted()` no necesita comparador. En el ejemplo la lista es de `String` y como sabemos `String` es `Comparable`

```
import java.util.Arrays;
import java.util.List;

public class App{
    public static void main(String[] args) throws Exception {
        List<String> letras = Arrays.asList("V","A","J");
        letras.stream().sorted().forEach(System.out::println);
    }
}
```

Para clases no comparables o para usar un orden diferente de comparable, pasamos un comparador

```
import java.util.Arrays;
import java.util.List;

class Persona{ //no implementa Comparable, por tanto no se puede usar sorted sin un Comparator
    String nombre;
    int edad;
    public Persona(String nombre, int edad) {
        this.nombre = nombre;
        this.edad = edad;
    }
    @Override
    public String toString() {
        return nombre + "-" + edad;
    }
}
```

```
public class App{
    public static void main(String[] args) {
        List<Persona> personas = Arrays.asList(new Persona("yo",22),new Persona("tu",15),new
Persona("el",50));
        personas.stream().sorted( (p1,p2)->p1.edad-p2.edad).forEach(System.out::println);
    }
}
```

Para obtener el orden inverso hay muchas posibilidades, nos contentamos con las siguientes posibilidades:

- si usamos orden natural podemos obtener inverso con `reverseOrder()`
- si usamos comparador con `reversed()`

El método `reverseOrder()` de Comparator

Si queremos el reverso del orden natural de clases que implementan `Comparable`, no tenemos que escribir el comparador, simplemente podemos usar el método static de `Comparator` `reverseOrder()`

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

public class App{
    public static void main(String[] args) throws Exception {
        List<String> letras = Arrays.asList("V","A","J");
        letras.stream().sorted(Comparator.reverseOrder()).forEach(System.out::println);
    }
}
```

El método `reversed()` de Comparator

Si consultas el API de `Comparator` observamos que `reversed()` es un método default, por tanto tiene ya una implementación por defecto lista para usar sobre un objeto comparador y que devuelve un nuevo comparador que impone el orden contrario al comparador original

Ejemplo

```
import java.util.Arrays;
import java.util.Comparator;
import java.util.List;
import java.util.stream.Collectors;

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

```

    }
    @Override
    public String toString() {
        return nombre + "-" + edad;
    }
}

public class App{

    public static void main(String[] args) {
        Comparator<Persona> miComparador=(p1,p2)->p1.edad-p2.edad;
        List<Persona> personas = Arrays.asList(new Persona("yo",22),new Persona("tu",15),new
Persona("el",50));

        List<Persona> personasOrdenadasPorEdad = personas.stream().sorted(
miComparador).collect(Collectors.toList());
        System.out.println(personasOrdenadasPorEdad);
        //ahora orden inverso al de miComparador
        personasOrdenadasPorEdad = personas.stream().sorted(
miComparador.reversed()).collect(Collectors.toList());
        System.out.println(personasOrdenadasPorEdad);

    }
}

```

El método `comparing()` de `Comparator`, un asistente para crear comparadores.

Si te fijas en el API, es un método que **devuelve un comparador**, es decir, es un método que nos ayuda a crear de forma breve un comparador

Cuando escribimos un comparador **es típico querer crear un comparador que se basa en "un campo clave"**, por ejemplo para nuestra clase `Persona` podemos querer ordenar por nombre o por edad. Este tipo de comparadores son tan comunes que se automatiza un poco la escritura de los comparadores simplemente **indicando "la clave" a `comparing()` y** él se encarga de construir el comparador correspondiente.

La clave se indica a `comparing` **utilizando una lambda o referencia a métodos.**

En el siguiente ejemplo observamos que `comparing` simplemente nos ahorra la lógica de la comparación. En este ejemplo lo hicimos con `p1.edad-p2.edad` que ya es un código muy breve y puede parecer que no merece la pena usar `comparing()`, pero en otras situaciones la expresión lambda sería más abultada y "ahí en el medio" molesta a la vista.

Observa que ya que se trata de ser breves este es un buen contexto para usar **referencias a métodos**, para eso necesitamos **métodos get()** para acceder a las propiedades.

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

class Persona{
    String nombre;
    int edad;
    public Persona(String nombre, int edad) {
        this.nombre = nombre;
        this.edad = edad;
    }
    public int getEdad() {
        return edad;
    }
    @Override
    public String toString() {
        return nombre + "-" + edad;
    }
}

public class App{
    public static void main(String[] args) {
        List<Persona> personas = Arrays.asList(new Persona("yo",22),new Persona("tu",15),new
Persona("el",50));
        personas.stream().sorted((p1,p2)->p1.edad-p2.edad).forEach(System.out::println);
        personas.stream().sorted(Comparator.comparing(p->p.edad)).forEach(System.out::println);
        personas.stream().sorted(Comparator.comparing(p->p.getEdad())).forEach(System.out::println);
        personas.stream().sorted(Comparator.comparing(Persona::getEdad)).forEach(System.out::println);
    }
}
```

comparing() + reversed()

Usando **comparing()** y **reversed()** tenemos cubierta la escritura "breve" de los comparadores más habituales que consisten en ordenar por un campo ascendente o descendientemente y todo ello con una escritura breve y concisa. Recuerda que la concisión es muy importante al escribir un stream para que sea legible.

```
List<Persona> personas = Arrays.asList(new Persona("yo",22),new Persona("tu",15),new Persona("el",50));
// asc
personas.stream().sorted((p1,p2)->p1.nombre.compareTo(p2.nombre)).forEach(System.out::println);
System.out.println("");
// desc
personas.stream().sorted(Comparator.comparing(Persona::getNombre).reversed()).forEach(System.out::println);
```

Más métodos implicados en la ordenación, ejemplo **thenComparing()**.

Hay más métodos cuyo objetivo es trabajar en un Stream de forma concisa y buscando el estilo de programación funcional donde cada función resuelve un pequeño problema. Vemos a continuación un ejemplo con theComparing().

Supongamos que queremos ordenar por **varios campos clave**, es decir, necesitamos usar **condiciones compuestas**, lo que implica escribir ifs más largos o bien usar recursos funcionales extra que en nuestro caso consiste en usar thenComparing()

En el siguiente ejemplo ordenamos **por edad y después "then" por nombre**.

```
List<Persona> personas = Arrays.asList(new Persona("yo",15),new Persona("tu",15),new Persona("el",50));
personas.stream().sorted(Comparator.comparing(Persona::getEdad).thenComparing(Persona::getNombre)).forEach(System.out::println);
```

EJERCICIO U8_B8_E1:

En este ejemplo, si observas el main renunciamos a utilizar los recursos de la programación funcional y pretendemos ordenar una lista de producto por tres campos: cantidad, precio y descripción. Ni siquiera usamos expresiones lambda para escribir el comparador con lo que tenemos un código "muy vertical"

```
import java.util.*;

class Producto {
    private String descripcion;
    private int cantidad;
    private double precio;

    public Producto(String descripcion, int cantidad, double precio) {
        this.descripcion = descripcion;
        this.cantidad = cantidad;
        this.precio = precio;
    }

    public String getDescripcion() {
        return descripcion;
    }

    public int getCantidad() {
        return cantidad;
    }

    public double getPrecio() {
        return precio;
    }
}

class App{

    public static void main(String[] args) {
        // Creamos una lista de productos desordenada
        List<Producto> lista = new ArrayList<>();
        lista.add(new Producto("Producto A", 10, 50.0));
        lista.add(new Producto("Producto B", 5, 100.0));
        lista.add(new Producto("Producto C", 20, 10.0));
        lista.add(new Producto("Producto A", 5, 20.0));
    }
}
```

```

lista.add(new Producto("Producto B", 5, 10.0));
lista.add(new Producto("Producto D", 5, 10.0));

// Creamos un comparador para ordenar primero por cantidad, luego por precio y finalmente por
descripción
Comparator<Producto> comparador = new Comparator<Producto>() {
    @Override
    public int compare(Producto p1, Producto p2) {
        int resultado = Integer.compare(p1.getCantidad(), p2.getCantidad());
        if (resultado != 0) {
            return resultado;
        }
        resultado = Double.compare(p1.getPrecio(), p2.getPrecio());
        if (resultado != 0) {
            return resultado;
        }
        return p1.getDescripcion().compareTo(p2.getDescripcion());
    }
};

// Ordenamos la lista utilizando el comparador
Collections.sort(lista, comparador);

// Imprimimos la lista ordenada
for (Producto p : lista) {
    System.out.println(p.getCantidad() + " - " + p.getPrecio() + " - " + p.getDescripcion());
}
}

```

Se pide un main escrito con estilo funcional “más horizontal”. Cuando consigas tu solución que tendrá un aspecto similar a los últimos ejemplos de **thencomparing()**, compara y reflexiona sobre los estilos. Busca una solución legible. Tampoco hay que obsesionarse con escribir todo en una línea pues la abundancia de paréntesis y llamadas a métodos puede ser nocivo para la legibilidad ¿O no?