FUNDAMENTOS DE PROGRAMACIÓN

6 – Interfaz Comparator

UNIVERSIDAD DE SEVILLA

ÍNDICE

ÍNDICE

1. Orden natural

- La interfaz del tipo T debe extender a la interfaz Comparable<T>, y en la clase hay que implementar el método compareTo

```
public class Vuelo implements Comparable<Vuelo> {...}
-----------------------------------------------------
public int compareTo(Vuelo v) {
    int res = getCodigo().compareTo(v.getCodigo());
    if (res == 0) {
        res = getFechaSalida().compareTo(v.getFechaSalida());
    }
    return res;
}
-----------------------------------------------------
List<Vuelo> vuelos = new LinkedList<Vuelo>();
...
Collections.sort(vuelos);
```

1. Orden Alternativo

- Muchas veces necesitamos ordenar por otros criterios distintos al orden natural. Por ejemplo, nos puede interesar ordenar vuelos por su destino, su fecha de salida o su porcentaje de ocupación.

```
Collections.sort(vuelos, orden alternativo por fecha de salida);
System.out.println("Vuelos por orden de fecha de salida: " + vuelos);
```

```
Comparator<Vuelo> comparadorVueloFechaSalida = new
ComparadorVueloFechaSalida();
Collections.sort(vuelos, comparadorVueloFechaSalida);
System.out.println("Vuelos por orden de fecha de salida: " + vuelos);
```

ÍNDICE

2. Interfaz Comparator

- La interfaz Comparator es una interfaz funcional, ya que mediante ella creamos un tipo que representa una función. En este caso, cuando construimos un objeto de tipo Comparator, lo que estamos creando es una función para comparar dos objetos. Y esta función se puede pasar como parámetro a ciertos métodos, como es el caso del método sort.
  - Este método espera como segundo parámetro un tipo funcional que le indique cómo debe ordenar los elementos de la lista. El objeto Comparator cumple con este requisito.

- A partir de Java 8 las interfaces pueden contener métodos de tipo static. Estos métodos se implementan en la propia interfaz, y pueden ser invocados desde las clases que implementan la interfaz.

- Otra novedad que tuvo Java 8 es que permitió incluir en una interfaz implementaciones por defecto (default) para los métodos. Si una clase que implementa la interfaz no redefine este método, utilizará la implementación por defecto.

2. Interfaz Comparator

- Pues bien, las interfaces funcionales son aquellas que sólo incluyen un método que no es de tipo static o default. Es el caso de la interfaz Comparator. Esta interfaz tiene, entre otros, los siguientes métodos:

```
public interface Comparator<T> {
    int compare(T o1, T o2);
    static <T,U extends Comparable<? super U>> Comparator<T> comparing(Function<?
super T,? extends U> keyExtractor);
    static <T extends Comparable<? super T>> Comparator<T> naturalOrder();
    default Comparator<T> reversed();
    static <T extends Comparable<? super T>> Comparator<T> reverseOrder();
    default Comparator<T> thenComparing(Comparator<? super T> other);
}
```

ÍNDICE

3. Comparadores.
Opción 1: Definición mediante una clase que implementa Comparator

- La opción clásica en Java es crear una clase que implemente la interfaz Comparator. Esta clase implementará el método compare de la interfaz
- Este método recibe dos objetos y compara los valores de la propiedad o propiedades según las cuales queremos ordenar los objetos. El valor devuelto es similar al devuelto por el método compareTo

```
public class ComparadorVueloFechaSalida implements Comparator<Vuelo> {
    public int compare(Vuelo v1, Vuelo v2) {
        int res = v1.getFechaSalida().compareTo(v2.getFechaSalida());
        return res;
    }
}
-----------------------------------------------------------------------
Comparator<Vuelo> comparadorVueloFechaSalida = new ComparadorVueloFechaSalida();

Collections.sort(vuelos, comparadorVueloFechaSalida);
System.out.println("Vuelos por orden de fecha de salida: " + vuelos);
```

3. Comparadores.
Opción 2: Definición mediante el método comparing

- La forma más simple de crear un objeto Comparator es indicar directamente la propiedad según la cual queremos ordenar los objetos.
- Para ello usamos un método de la interfaz Comparator, el método **`comparing`**.
- En el ejemplo de la ordenación de vuelos por su fecha de salida se haría de la siguiente manera:

```
Comparator<Vuelo> comparadorVueloFechaSalida = Comparator.comparing(Vuelo::getFechaSalida);

Collections.sort(vuelos, comparadorVueloFechaSalida);
System.out.println("Vuelos por orden de fecha de salida: " + vuelos);
```

3. Comparadores.
Opción 2: Definición mediante el método comparing

- La expresión que indica el orden que deseamos tiene la forma:
  - <Nombre del tipo>::<Nombre del método>

  , siendo el método el getter que devuelve la propiedad por la cual queremos ordenar.

```
Vuelo::getFechaSalida
```

```
Comparator<Vuelo> comparadorVueloFechaSalida = Comparator.comparing(Vuelo::getFechaSalida);
```

3. Comparadores.
Opción 2: Definición mediante el método comparing


```
Comparator<Vuelo> comparadorVueloOcupacion =
        Comparator.comparing(x -> 100. * x.getNumPasajeros() / x.getNumPlazas());

Collections.sort(vuelos, comparadorVueloOcupacion);
```

- **Expresión lambda:** simplificación de un método en el que el parámetro de entrada y la expresión que produce el valor de salida se separan por el operador flecha '->'. Sería equivalente al método:

```
public Double getPorcentajeOcupacion() {
   return 100. * getNumPasajeros() / getNumPlazas();
}
```

3. Comparadores.
Opción 3: Definición mediante expresiones lambda

- Una tercera forma de definir el comparador es utilizar directamente una **expresión lambda**, sin necesidad de recurrir al método comparing. Para el ejemplo de ordenación por la fecha de salida del vuelo, sería de la siguiente forma:

```
Comparator<Vuelo> comparadorVueloFechaSalida =
      (x, y) -> x.getFechaSalida().compareTo(y.getFechaSalida());

Collections.sort(vuelos, comparadorVueloFechaSalida);
System.out.println("Vuelos por orden de fecha de salida: " + vuelos);
```

3. Combinación de comparadores

- En ocasiones nos interesará ordenar según un criterio y, a igualdad de éste, según un segundo criterio.
- Por ejemplo, queremos ordenar los vuelos por fecha de salida, y a igualdad de ésta por su destino. Para ello creamos dos objetos de tipo comparator y los combinamos usando el método **thenComparing** de la interfaz Comparator:

```
Comparator<Vuelo> comparadorVueloFechaSalida = Comparator.comparing(Vuelo::getFechaSalida);
Comparator<Vuelo> comparadorVueloDestino = Comparator.comparing(Vuelo::getDestino);

Collections.sort(vuelos,comparadorVueloFechaSalida.thenComparing(comparadorVueloDestino));
```

3. Ordenación inversa

- ¿Qué haríamos si queremos obtener una lista en orden inverso?, por ejemplo, de mayor a menor número de pasajeros. Pues utilizamos el método `reversed()` de la interfaz Comparator:

```
Comparator<Vuelo> comparadorVueloNumPasajeros = Comparator.comparing(Vuelo::getNumPasajeros);

Collections.sort(vuelos, comparadorVueloNumPasajeros.reversed());
```

ÍNDICE

4. Ejemplos

- Ejemplo 1. Obtener el vuelo con más pasajeros:

```
Vuelo masPasajeros = Collections.max(vuelos,
    Comparator.comparing(Vuelo::getNumPasajeros));
System.out.println("Vuelo con más pasajeros: " +
masPasajeros);
```

- El método max obtiene el mayor elemento de una colección según el orden indicado por el comparador que recibe como parámetro. En este caso, nos da el vuelo de la lista con mayor número de pasajeros.

4. Ejemplos

- Ejemplo 2. Obtener el vuelo que sale antes:

```
Vuelo primerVuelo = Collections.min(vuelos,
    Comparator.comparing(Vuelo::getFechaSalida));
System.out.println("Vuelo que sale antes: " +
primerVuelo);
```

4. Ejemplos

- Ejemplo 3. Crear un conjunto de vuelos ordenado por el número de pasajeros:

```
SortedSet<Vuelo> vuelosOrdenados =
    new TreeSet<Vuelo>(Comparator.comparing(Vuelo::getNumPasajeros));
```

- En este ejemplo se nos plantea un problema: dos vuelos con el mismo número de pasajeros son iguales según nuestro orden, y por lo tanto no se podrían añadir los dos elementos al SortedSet. Para solucionar esto es preciso recurrir al orden natural del tipo en caso de empate por el orden alternativo. Para ello tenemos el método naturalOrder de la interfaz Comparator, y lo usaríamos de la siguiente forma:

```
SortedSet<Vuelo> vuelosOrdenados =
    new TreeSet<Vuelo>(Comparator.comparing(Vuelo::getNumPasajeros)
        .thenComparing(Comparator.naturalOrder()));
```

4. Ejemplos

- Ejemplo 4. Crear un SortedMap que relacione los vuelos con su número de pasajeros, ordenado por el destino del vuelo y a igualdad de éste por su orden natural.

```
SortedMap<Vuelo, Integer> pasajerosVuelos =
    new TreeMap<Vuelo, Integer>
(Comparator.comparing(Vuelo::getDestino)
            .thenComparing(Comparator.naturalOrder()));
```