Módulo profesional Programación

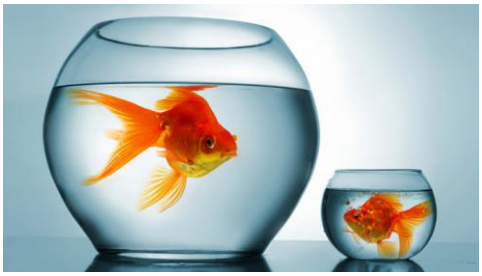
UD 8.1.1 – Ordenar colecciones de objetos

Contenido

Interfaces para **ordenar objetos** de clases definidas por el usuario:

- Comparator
- Comparable

¿Cómo comparamos dos objetos en Java?



Pero primero, esta pregunta:

¿**Por qué** puede surgir la necesidad de comparar un objeto Java con otro?

Comparando objetos en Java

La aplicación más importante para comparar dos objetos es, sin duda, la **ordenación de colecciones de objetos.**

Para ordenar objetos, el programa tiene que compararlos y averiguar si un objeto es más pequeño, más grande o igual que otro.

Comparando objetos en Java

Los tipos de datos primitivos en Java (int, long, double, etc.) se comparan con los operadores

<, <=, ==, =>, >, y !=.

Esto **no funciona** con los objetos.

Comparando objetos en Java

Las comparaciones en Java son bastante fáciles hasta que dejan de serlo…

Comparando objetos en Java

¿Cómo comparamos dos objetos String en Java?

Comparando objetos en Java

Supongamos que tenemos las siguientes cadenas:

```
String c1 = "Felices";
String c2 = "Programadores";
```

Ahora queremos determinar si c1 es menor, mayor o igual que c2. En otras palabras: si -según la ordenación alfabética- c1 estaría antes o después de c2.

Lo hacemos de la siguiente manera:

```
int resultado = c1.compareTo(c2);
```

Comparando objetos en Java

```
int resultado = c1.compareTo(c2);
```

La variable de resultado contiene ahora:

- un valor **menor que 0** si c1 es anterior a c2 según la ordenación alfabética
- **0**, si c1 y c2 son iguales (es decir, c1.equals(c2) es verdadero)
- un valor **mayor que 0** si c1 viene después de c2 según la ordenación alfabética.

En el ejemplo anterior, el resultado sería menor que 0 porque “Felices" se ordenaría antes de “Programadores".

Comparando objetos en Java

El método **compareTo()** se utiliza "entre bambalinas" para ordenar un array o una lista de objetos, por ejemplo, de la siguiente manera:

Nota: "entre bambalinas"→ cuando nos referimos a llevar a cabo una acción de modo reservado, discretamente, sin **que** se enteren los demás.

Ordenar listas de objetos en Java

```
public class OrdenarListaCadenas{
    public static void main(String [] args)
    {
        List<String> lista = new ArrayList<>();

        lista.add("Maleta");
        lista.add("Muñequera");
        lista.add("Póster");
        lista.add("Tarjeta");
        lista.add("Post-it");
        lista.add("Teléfono");
        lista.add("Lámpara");
        lista.add("Auriculares");

        Collections.sort(lista);
        System.out.println(lista);

    }
}
```

```
[Auriculares, Lámpara, Maleta, Muñequera, Post-it, Póster, Tarjeta, Teléfono]
```

Ordenar listas de objetos en Java

Por ejemplo, si queremos ordenar en orden inverso podemos hacer:

Collections.sort(lista,Collections.reverseOrder());

o

Collections.sort(lista);

Collections.reverse(lista);

```
[Auriculares, Lámpara, Maleta, Muñequera, Post-it, Póster, Tarjeta, Teléfono]
[Teléfono, Tarjeta, Póster, Post-it, Muñequera, Maleta, Lámpara, Auriculares]
[Auriculares, Lámpara, Maleta, Muñequera, Post-it, Póster, Tarjeta, Teléfono]
[Teléfono, Tarjeta, Póster, Post-it, Muñequera, Maleta, Lámpara, Auriculares]
```

```java
public class OrdenarListaCadenas{
    public static void main(String [] args)
    {
        List<String> lista = new ArrayList<>();

        lista.add("Maleta");
        lista.add("Muñequera");
        lista.add("Póster");
        lista.add("Tarjeta");
        lista.add("Post-it");
        lista.add("Teléfono");
        lista.add("Lámpara");
        lista.add("Auriculares");

        Collections.sort(lista);
        System.out.println(lista);

        Collections.sort(lista, Collections.reverseOrder());
        System.out.println(lista);

        Collections.sort(lista);
        System.out.println(lista);

        Collections.reverse(lista);
        System.out.println(lista);

    }
}
```

Comparando objetos en Java

El método **compareTo()** de la clase String que hemos utilizado anteriormente se encuentra declarado en la interfaz **Comparable**, implementada por la clase **String**.

La interfaz Comparable define únicamente este método. Todas las clases cuyos objetos deben ser comparables lo implementan. Además de **String**, también lo implementan por ejemplo, **Integer, Long, Date, LocalDateTime**, y muchas más.

Comparando objetos en Java

El orden resultante del método **compareTo()** se denomina "orden natural": Las cadenas se ordenan alfabéticamente; las fechas y horas se ordenan en orden cronológico, etc.

Comparando objetos en Java

¿Cómo comparamos dos objetos en Java

de clases definidas por el usuario?

Ordenar listas en Java

```java
public class Jugador {
    private String nombre;
    private int posicion;
    private int goles;

    //...
}
```

Ahora bien, si tenemos por ejemplo, los jugadores de un equipo de fútbol representados como objetos Jugador, aunque cuando llamemos al método **sort**, la clase **Collections** no va a tener ni idea de cómo ordenar dos jugadores de fútbol.

En el ejemplo solo tienen datos de nombre, posición y goles y lo que queremos es ordenarlos por posición, y si la posición es la misma, por nombre.

Ordenar listas en java

Java ofrece dos interfaces que serán claves para realizar un ordenamiento simple, las cuales son **Comparator** y **Comparable.**

Estas dos interfaces atienden contextos diferentes en el ordenamiento de las listas, pues existen **dos escenarios concretos:**

- **Objetos** que preparamos para que puedan ser ordenados
- **Objetos** no preparados para ser ordenados y proveemos una clase externa para ello.

Objetos que preparamos para ser ordenados

El escenario más simple para ordenar un Lista, es cuando sus elementos **están preparados para ser ordenado**s, esto implica que los elementos de la lista **implementan la interface Comparable.**

```
public interface Comparable<T> {
    public int compareTo(T otro);
}
```

Al implementar esta interface, la clase de la cual los objetos de son instancias, deberá implementar el método

public int compareTo(T o)

Objetos que preparamos para ser ordenados

Este método sirve para comprar un objeto contra otro, recibe como parámetro otro objeto, el cual utilizaremos para comprarlo contra el actual para responder con un entero, que deberá ser:

- < 0 (Menor que cero): cuando el objeto actual es menor que el otro
- = 0 (Igual a cero): cuando los objetos son iguales
- > 0 (Mayor que cero): cuando el otro objeto es mayor.

Objetos que preparamos para ser ordenados

Cuando una clase implementa **Comparable** entonces podemos decir que está preparada para ser ordenada y ordenarla en una lista será tan simple como hacer lo siguiente:

Collections.sort(lista);

Objetos que preparamos para ser ordenados

```java
public class Jugador implements Comparable<Jugador>
{
    private String nombre;
    private int posicion;
    private int goles;

    public static final int PORTERO = 0;
    public static final int DEFENSA = 1;
    public static final int CENTROCAMPISTA = 2;
    public static final int DELANTERO = 3;

    private String[] posiciones = new String[] {"Portero","Defensa", "Centrocampista", "Delantero"};

    public Jugador(String nombre, int posicion, int goles){
        this.nombre = nombre;
        this.posicion=posicion;
        this.goles=goles;
    }

    public String toString(){
        return nombre+": "+this.posiciones[posicion]+" ha marcado "+goles+" goles";
    }

    public int compareTo(Jugador j){
        if (this.posicion==j.posicion)
            return this.nombre.compareTo(j.nombre);
        else
            return this.posicion-j.posicion;
    }

    public int getPosicion(){
        return this.posicion;
    }

    public String getNombre(){
        return this.nombre;
    }
}
```

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

public class OrdenarJugador
{
    public static void main(String [] args)
    {
        List<Jugador> lista = new ArrayList<>();

        lista.add(new Jugador("Iker Casillas", Jugador.PORTERO,3));
        lista.add(new Jugador("Jesús Navas", Jugador.DELANTERO,10));
        lista.add(new Jugador("Xabi Alonso", Jugador.CENTROCAMPISTA,4));
        lista.add(new Jugador("Sergio Ramos", Jugador.DEFENSA,2));
        lista.add(new Jugador("Álvaro Arbeloa", Jugador.DEFENSA,1));
        lista.add(new Jugador("Andrés Iniesta", Jugador.CENTROCAMPISTA, 12));
        lista.add(new Jugador("David Villa", Jugador.DELANTERO,15));
        lista.add(new Jugador("Fernando Torres", Jugador.DELANTERO,2));
        lista.add(new Jugador("José Manuel Reina", Jugador.PORTERO, 1));
        lista.add(new Jugador("Roberto Soldado", Jugador.DELANTERO,12));
        lista.add(new Jugador("Santiago Cazorla", Jugador.CENTROCAMPISTA,5));
        lista.add(new Jugador("Jordi Alba", Jugador.DEFENSA,2));
        lista.add(new Jugador("Cesc Fábregas", Jugador.CENTROCAMPISTA,4));

        Collections.sort(lista);
        lista.forEach((n) -> System.out.println(n));

    }
}
```

```
Iker Casillas: Portero ha marcado 3 goles
José Manuel Reina: Portero ha marcado 1 goles
Jordi Alba: Defensa ha marcado 2 goles
Sergio Ramos: Defensa ha marcado 2 goles
Álvaro Arbeloa: Defensa ha marcado 1 goles
Andrés Iniesta: Centrocampista ha marcado 12 goles
Cesc Fábregas: Centrocampista ha marcado 4 goles
Santiago Cazorla: Centrocampista ha marcado 5 goles
Xabi Alonso: Centrocampista ha marcado 4 goles
David Villa: Delantero ha marcado 15 goles
Fernando Torres: Delantero ha marcado 2 goles
Jesús Navas: Delantero ha marcado 10 goles
Roberto Soldado: Delantero ha marcado 12 goles
```

Actividad

- Ordenar colecciones de objetos **Circulo** por el atributo radio (int), empleando la interfaz Comparable

Objetos no preparados para ser ordenados y proveemos una clase externa para ello.

Por otra parte, las clases que no implementan Comparable no podrán ser ordenadas de forma natural, por lo que tendremos que recurrir a la interface **Comparator**, la cual sirve para crear una clase externa que ayude al ordenamiento de los objetos sin modificar la estructura de las clases existentes.

Para lo cual será necesario **crear una nueva clase** que implementa la interface **Comparator** y con ello el método

```
public int compare(Object obj1, Object obj2):
```

Objetos no preparados para ser ordenados y proveemos una clase externa para ello.

Este método, **compare**, funciona exactamente igual que el de la interface Comparable, solo que este recibe como parámetros los dos objetos que será comparados.

El tipo **Comparator** está definido en el **paquete java.util** y se compone solo del método compare.

Para el ejemplo de los jugadores

```
import java.util.Comparator;

class ComparadorJugadores implements Comparator<Jugador2>{

   public int compare(Jugador2 j1,Jugador2 j2){
        return j1.getNombre().compareToIgnoreCase(j2.getNombre());
   }

}
```

Objetos no preparados para ser ordenados y proveemos una clase externa para ello.

Ahora bien, si queremos ordenar una lista mediante este objeto podríamos hacerlo de la siguiente manera:

```
Collections.sort(lista, new ComparadorJugadores());
```

Donde lista es la lista de objetos a ordenar y ComparadorJugadores es una clase que implementa la interfaz Comparator.

```java
public class Jugador2 {

    private String nombre;
    private int posicion;
    private int goles;

    public static final int PORTERO = 0;
    public static final int DEFENSA = 1;
    public static final int CENTROCAMPISTA = 2;
    public static final int DELANTERO = 3;

    private String[] posiciones = {"Portero","Defensa", "Centrocampista", "Delantero"};

    public Jugador2(String nombre, int posicion, int goles){
        this.nombre = nombre;
        this.posicion=posicion;
        this.goles=goles;
    }

    public String toString(){
        return nombre+": "+this.posiciones[posicion]+" ha marcado "+goles+" goles";
    }

    public int getPosicion(){
            return this.posicion;
    }

    public String getNombre(){
            return this.nombre;
    }
}
```

```java
import java.util.Comparator;

class ComparadorJugadores implements Comparator<Jugador2>{

   public int compare(Jugador2 j1,Jugador2 j2){
        return j1.getNombre().compareToIgnoreCase(j2.getNombre());
   }

}
```

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

public class OrdenarJugador2 {

    public static void main(String[] args) {

        List<Jugador2> lista = new ArrayList<>();

        lista.add(new Jugador2("Iker Casillas", Jugador2.PORTERO,3));
        lista.add(new Jugador2("Jesús Navas", Jugador2.DELANTERO,10));
        lista.add(new Jugador2("Xabi Alonso", Jugador2.CENTROCAMPISTA,4));
        lista.add(new Jugador2("Sergio Ramos", Jugador2.DEFENSA,2));
        lista.add(new Jugador2("Álvaro Arbeloa", Jugador2.DEFENSA,1));
        lista.add(new Jugador2("Andrés Iniesta", Jugador2.CENTROCAMPISTA, 12));
        lista.add(new Jugador2("David Villa", Jugador2.DELANTERO,15));
        lista.add(new Jugador2("Fernando Torres", Jugador2.DELANTERO,2));
        lista.add(new Jugador2("José Manuel Reina", Jugador2.PORTERO, 1));
        lista.add(new Jugador2("Roberto Soldado", Jugador2.DELANTERO,12));
        lista.add(new Jugador2("Santiago Cazorla", Jugador2.CENTROCAMPISTA,5));
        lista.add(new Jugador2("Jordi Alba", Jugador2.DEFENSA,2));
        lista.add(new Jugador2("Cesc Fábregas", Jugador2.CENTROCAMPISTA,4));

        Collections.sort(lista, new ComparadorJugadores());

        lista.forEach((n) -> System.out.println(n));
    }
}
```

Objetos no preparados para ser ordenados y proveemos una clase externa para ello.

Aunque también podríamos **ahorrarnos la creación de una nueva clase y crear una clase anónima,** como podemos ver a continuación:

```java
Collections.sort(lista, new Comparator<Jugador2>(){
    @Override
    public int compare(Jugador2 j1, Jugador2 j2) {
        return j1.getNombre().compareToIgnoreCase(j2.getNombre());
    }
});
```

Veamos que este último ejemplo instanciamos la interface Comparator al vuelo y definimos el método compare para ordenar jugadores.

Objetos no preparados para ser ordenados y proveemos una clase externa para ello.

Finalmente, Java 8 ofrece una forma muchos más simple de ordenar los objetos utilizado **lamba expresión**, por lo cual en lugar de crear un **Comparator**, solo definimos la estrategia para comparar los objetos, veamos cómo quedaría:

```
Collections.sort(lista, (x, y) -> x.getNombre().compareToIgnoreCase(y.getNombre()));
```

Actividad

Ordenar colecciones de objetos Empleado por:

- Edad o
- Salario o
- Nombre

empleando la interfaz Comparator.

Comparable vs Comparator

Comparable	Comparator
Comparable proporciona una única secuencia de ordenación. En otras palabras, podemos ordenar la colección basándonos en un solo elemento como el dni, el nombre, etc.	Comparator proporciona múltiples secuencias de ordenación. En otras palabras, podemos ordenar la colección en base a múltiples elementos como el dni, nombre, etc.
Comparable afecta a la clase original, lo que significa que la clase actual se modifica.	Comparator no afecta a la clase original, lo que significa que la clase actual no se modifica.
Comparable proporciona el método compareTo() para ordenar los elementos.	Comparator proporciona el método compare() para ordenar los elementos.
Comparable está en el paquete java.lang.	El comparador está en el paquete java.util.
Podemos ordenar los elementos de la lista mediante el método Collections.sort(List).	Podemos ordenar los elementos de la lista mediante el método Collections.sort(List, Comparator).

Colecciones y su ordenación

Una clase puede especificar una relación de orden por medio de:

- La interfaz Comparable **(orden natural)**
- La interfaz Comparator **(orden alternativo)**

Los objetos que implementan un orden natural o alternativo pueden ser utilizados:

- Como elementos de un **conjunto ordenado** (TreeSet).
- Como claves en un **mapa ordenado** (TreeMap).
- En listas ordenables con el método **sort(...)**, etc.

Un ejemplo práctico

Pasemos mejor a un ejemplo práctico. Vamos a ver un ejemplo rápido para realizar los tipos de ordenamientos que vimos anteriormente.

Actividad

Implementa una clase de **Movie** que tenga variables miembro **rating**, **name** y **year**.

Supongamos que deseamos ordenar una lista de películas según el año de lanzamiento (year).

La clase Movie debe implementar la interfaz Comparable y el método compareTo()

Prueba la clase **Movie** con la siguiente clase. La salida deberá ser:

```
Movies after sorting :
Star Wars 8.7 1977
Empire Strikes Back 8.8 1980
Return of the Jedi 8.4 1983
Force Awakens 8.3 2015
```

```
import java.util.List;
import java.util.ArrayList;
class MoviSort {
    public static void main(String[] args)
    {
        List<Movie> list = new ArrayList<>();

        list.add(new Movie("Force Awakens", 8.3, 2015));
        list.add(new Movie("Star Wars", 8.7, 1977));
        list.add(new Movie("Empire Strikes Back", 8.8, 1980));
        list.add(new Movie("Return of the Jedi", 8.4, 1983));

        //Inserta el codigo para ordenar la lista

        System.out.println("Movies after sorting : ");
        for (Movie movie: list) {
            System.out.println(movie.getName() + " " +
                               movie.getRating() + " " +
                               movie.getYear());
        }
    }
}
```

Actividad

```
import java.io.*;
import java.util.*;

class Movie implements Comparable<Movie> {
    private double rating;
    private String name;
    private int year;

    public Movie(String nm, double rt, int yr) {
        this.name = nm;
        this.rating = rt;
        this.year = yr;
    }

    public double getRating() { return rating; }
    public String getName()   {  return name; }
    public int getYear()      {  return year;  }

    public int compareTo(Movie m) {
        return this.year - m.year;
    }
}
```

Actividad

Ahora, supongamos que queremos además ordenar las películas por sus atributos rating y name.

Cuando hacemos que un elemento de colección sea comparable (al tenerlo implementado como Comparable), solo tenemos una oportunidad de implementar el método compareTo(). La solución está utilizando **Comparator**.

Actividad

A diferencia de Comparable, la interfaz Comparator es externa al tipo de elemento que estamos comparando. Es una clase separada.

Creamos múltiples clases separadas (que implementan Comparator) para comparar por diferentes atributos.

Crea las clases **RatingCompare** y **NameCompare**

Actividad

```java
// Class to compare Movies by name
import java.util.*;
class NameCompare implements Comparator<Movie>
{
    public int compare(Movie m1, Movie m2)
    {
        return m1.getName().compareTo(m2.getName());
    }
}
```

```java
// Class to compare Movies by ratings
import java.util.*;
class RatingCompare implements Comparator<Movie>
{
    public int compare(Movie m1, Movie m2)
    {
        if (m1.getRating() < m2.getRating()) return -1;
        else if (m1.getRating() > m2.getRating()) return 1;
        else return 0;
    }
}
```

Actividad

4. Utiliza la siguiente clase de prue

La salida deberá ser

```
Sorted by rating
8.3 Force Awakens 2015
8.4 Return of the Jedi 1983
8.7 Star Wars 1977
8.8 Empire Strikes Back 1980

Sorted by name
Empire Strikes Back 8.8 1980
Force Awakens 8.3 2015
Return of the Jedi 8.4 1983
Star Wars 8.7 1977

Sorted by year
1977 8.7 Star Wars
1980 8.8 Empire Strikes Back
1983 8.4 Return of the Jedi
2015 8.3 Force Awakens
```

```java
import java.util.List;
import java.util.Collections;

class MovieSortTest
{
    public static void main(String[] args)
    {
        List<Movie> list = new ArrayList<>();
        list.add(new Movie("Force Awakens", 8.3, 2015));
        list.add(new Movie("Star Wars", 8.7, 1977));
        list.add(new Movie("Empire Strikes Back", 8.8, 1980));

        System.out.println("Sorted by rating");
        //Inserta el codigo para comparar por el atributo rating


        for (Movie movie: list)
            System.out.println(movie.getRating() + " " +
                               movie.getName() + " " +
                               movie.getYear());

        System.out.println("\nSorted by name");
        //Inserta el codigo para comparar por el atributo name


        for (Movie movie: list)
            System.out.println(movie.getName() + " " +
                               movie.getRating() + " " +
                               movie.getYear());

        // Usa comparable para ordenar por el atributo year
        System.out.println("\nSorted by year");
        Collections.sort(list);

        for (Movie movie: list)
            System.out.println(movie.getYear() + " " +
                               movie.getRating() + " " +
                               movie.getName()+" ");
    }
}
```

Resumen

La interfaz **Comparable** está pensado para objetos con **orden natural**, lo que significa que **el objeto en sí mismo debe saber cómo debe ordenarse**.

Lógicamente, la interfaz de **Comparable** compara "esta" referencia con el objeto especificado como parámetro (método compareTo) y **Comparator** en Java compara dos objetos diferentes proporcionados como parámetros (método compare).

Si alguna clase implementa una interfaz Comparable en Java, la colección de ese objeto, ya sea lista o vector, puede ordenarse automáticamente mediante el método Collections.sort() o Arrays.sort() y los objetos se ordenarán según el orden natural definido por el método **compareTo**.

Bibliografía y recursos web

- Curso youtube : Java desde 0
- Libro Java 9. Manual imprescindible. F. Javier Moldes Teo. Editorial Anaya
- App SoloLearn: Aprende a Programar. Curso Java
- Colecciones e iteradores – Universidad de Málaga