

Colecciones

Resumen realizado por Cristina Moreno Ruiz



Introducción.....	1
JAVA API.....	2
Enumerados.....	2
Collections Framework.....	4
Collection <E>.....	5
Iterator <E>.....	5
List.....	6
ArrayList.....	7
LinkedList.....	8
Map.....	9
HashMap.....	9
Set.....	9
HashSet.....	9
Otras colecciones.....	9
Comparando colecciones de objetos.....	10
Comparable<T>.....	10
Comparator <T>.....	12
Bibliografía.....	14

Introducción

Recordemos que Java tiene tipos primitivos y no primitivos.

Lo primitivos solo tienen definido su tamaño y tipo de valores que almacenaría una variable de ese tipo, como:

Tipo de datos	Tamaño (bytes)	Descripción
byte	1	Almacena números de -128 a 127
short	2	Almacena números de -32,768 a 32,767
int	4	Almacena números de -2^{31} a $2^{32}-1$ (-2,147,483,648 a 2,147,483,647)
long	8	Almacena números de -2^{63} a $2^{63}-1$
float	4	Almacena números fraccionarios con aprox 6 o 7 decimales
double	8	Almacena números fraccionarios con aprox 15 decimales
boolean	1 bit	Almacena valor verdadero o falso
char	2	Almacena un carácter, letra o valor 16 bit Unicode ¹

¹ Según [un tutorial](#) de Oracle, un char equivale a un “single 16-bit Unicode character”. No me queda claro si es UTF16 u otro estándar.

Puedes ver ejemplos en: https://www.w3schools.com/java/java_data_types.asp

Entre los tipos no primitivos que ya hemos utilizado: [Integer](#) y `String`. Son clases que contienen un valor de tipo primitivo y le añaden funcionalidad.

Java nos ofrece tipos de datos que hasta ahora no hemos utilizado.

Por ejemplo el tipo enumerado y otras clases predefinidas que tendremos que importar, para poder utilizar.

JAVA API

Estas clases y tipos de datos predefinidos, están disponibles en la API de Java.

Es una librería de clases preescritas, libres, que pueden utilizarse en JDE.

El listado de componentes puede consultarse en: <https://docs.oracle.com/javase/8/docs/api/>

Está organizada en paquetes y clases, de tal manera que puedes importar una clase, junto con sus métodos y atributos, o un paquete completo, con todas las clases que contiene dicho paquete.

```
import package.name.Class; // Import a single class
```

```
import java.util.Scanner;
```

```
import package.name.*; // Import the whole package
```

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

Puedes leer más sobre paquetes y clases en:

https://www.w3schools.com/java/java_packages.asp

Java utiliza la siguiente estructura en el sistema de archivos, para almacenar tus paquetes:

```
└── root
    └── mypack
        └── MyPackageClass.java
```

Enumerados

Existe una **clase** especial que representa un grupo de constantes. Para utilizar tipos enumerados, se usa la palabra reservada `enum`.

```
//defino el tipo enumerado Mes
```

```
public enum Mes { ENERO, FEBRERO, MARZO, ABRIL, MAYO, JUNIO,  
JULIO, AGOSTO, SEPTIEMBRE, OCTUBRE, NOVIEMBRE, DICIEMBRE };
```

No pueden añadirse elementos ni modificarlos, se comportan como *final*.

Una variable de tipo Mes, solo podrá tomar los valores enumerados definidos.

```
//Creo variables del tipo enumerado Mes
```

```
Mes mes;
```

```
Mes miMes = Mes.MARZO;
```

Ejemplo de W3Schools:

```
public class Main {  
    enum Level {  
        LOW,  
        MEDIUM,  
        HIGH  
    }  
  
    public static void main(String[] args) {  
        Level myVar = Level.MEDIUM;  
        System.out.println(myVar);  
    }  
}
```

Uso de Mes:

```
7  /**
8  *
9  * @author cristina
10 */
11 public class Meses {
12     //defino el tipo enumerado Mes
13     public enum Mes {
14         ENERO, FEBRERO, MARZO, ABRIL, MAYO, JUNIO, JULIO, AGOSTO, SEPTIEMBRE,
15         OCTUBRE, NOVIEMBRE, DICIEMBRE
16     };
17     //Método que recibe un mes como argumento y lo imprime
18     public static void imprimeMes(Mes otroMes){
19         System.out.println("Me voy a la playa en " +otroMes);
20     }
21
22     public static void main(String[] args) {
23         //Creo variables del tipo enumerado Mes
24         Mes mes;
25         Mes miMes = Mes.MARZO;
26
27         switch (miMes) {
28             case DICIEMBRE:
29             case ENERO:
30             case FEBRERO:
31                 System.out.println("Estoy en invierno");
32                 break;
33             case MARZO:
34                 System.out.println("Se nos acaba el mes");
35                 break;
36             case ABRIL:
37                 System.out.println("De vacaciones");
38                 break;
39             default:
40                 System.out.println("Cerca del verano");
41                 break;
42         }
43         imprimeMes(Mes.AGOSTO);
44     }
45 }
46
```

Características de los enumerados:

- Pueden recorrerse con un bucle, utilizando el método `.values()` de `enum`
- Utilizarse en `switch`
- Como clase está definida como `public`, `static` and `final`, así que no puede modificarse.
- No puede utilizarse para crear objetos
- No hay que importar la librería

Collections Framework

Es la estructura/arquitectura para representar y trabajar con colecciones en Java. Es muy amplia y no necesitamos conocerla al completo:

Collection <E>

Una colección es una clase interfaz, quiere decir que otras clases se encargarán de implementar sus métodos.

Una colección representa a un grupo de elementos de tipo <E>. Hay colecciones ordenadas, desordenadas, algunas permiten tener elementos repetidos otras no, ...

Según la documentación de Java, podemos hablar de los siguientes grupos de interfaces de colecciones:

- las Colecciones en sí misma: conjuntos, listas, colas y dobles colas (Double-Ended QUEue),
- y los mapas.

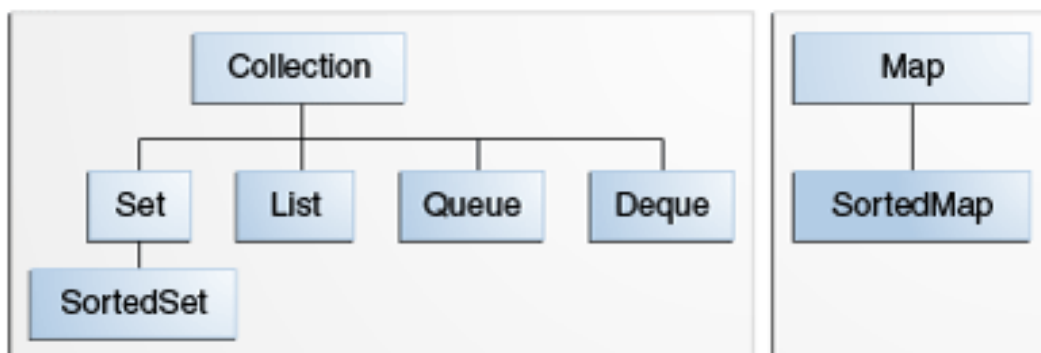


Figura 1: Interfaces principales de las colecciones (Fuente: <https://docs.oracle.com/javase/tutorial/collections/interfaces/index.html>)

Todas las colecciones tienen un constructor sin argumentos para crear una colección vacía o un constructor que recibe otra colección como argumento y la nueva colección tendrá todos los elementos de la que recibe como argumento el constructor.

Iterator <E>

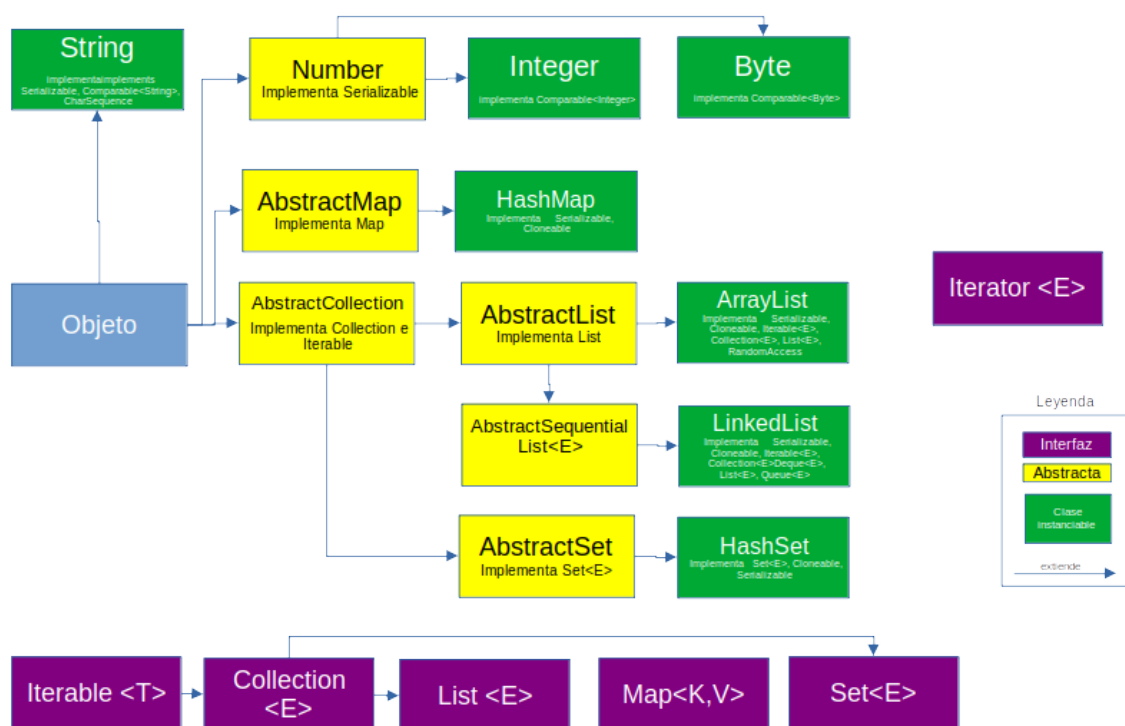
Es una clase interfaz que nos sirve como herramienta para movernos a través de los elementos de una colección de objetos de clase <E>. Cuando le pidamos el siguiente elemento, nos devolverá un objeto de clase <E>.

Observa sus métodos en <https://docs.oracle.com/javase/8/docs/api/java/util/Iterator.html> Tiene un método interesante, para aplicar una acción a los objetos que todavía no ha recorrido.

Existen interfaces que extienden Iterador, como [ListIterator](#), que puede moverse por una lista en ambas direcciones.

La clase Collection tiene un método `Iterator<E> iterator()`, que devuelve un iterador para recorrer la colección.

Para tener una idea de la relación entre las clases de colecciones que vamos a utilizar, puedes fijarte en este diagrama:



List

Es una interfaz que extiende a Collection y está ordenada, estando cada elemento en una posición *i*. **En resumen: una colección ordenada.**

El usuario tendrá control para saber en qué lugar inserta un nuevo elemento, consultar un elemento en concreto...

Ofrece varios métodos para buscar un objeto en concreto o múltiples objetos, en todo la lista o

desde una posición establecida...

Podemos revisar sus métodos en: <https://docs.oracle.com/javase/8/docs/api/java/util/List.html>

Algunas de las clases que implementan List y nos permitirán trabajar con colecciones de objetos son ArrayList y LinkedList.

ArrayList

Ya la hemos utilizado pero nos ayudará a entender otras clases que también son Colecciones de elementos.

Un [ArrayList](#) implementa la clase List y por lo tanto es una colección ordenada, simplemente es un array que puede redimensionarse y que tiene las propiedades de una Lista.

Fíjate qué tipo de clase son sus superclases:

[java.lang.Object](#)

[java.util.AbstractCollection](#)<E>

[java.util.AbstractList](#)<E>

[java.util.ArrayList](#)<E>

Al extender AbstractList, permite el acceso no secuencial, a cualquiera de sus elementos. Además implementa Serializable, Cloneable, Iterable, Collection, RandomAccess: así que puede iterarse, clonarse, accederse aleatoriamente...

La clase ArrayList incluye sus propios métodos para iterar y no necesita un iterador externo: get(int), set(int, E), add(int, E) and remove(int).

El método para aplicar una acción a todos sus elementos es:

```
void    forEach(Consumer<? super E> action)
```

Ojo que si varios programas, acceden concurrentemente a un ArrayList, añadiendo, eliminando o modificando elementos, podrían no estar accediendo a elementos sincronizados.

Podemos crear un ArrayList vacío o con los elementos de otro ArrayList. Lo vemos en el siguiente ejemplo, donde usamos una clase Escritora.

```

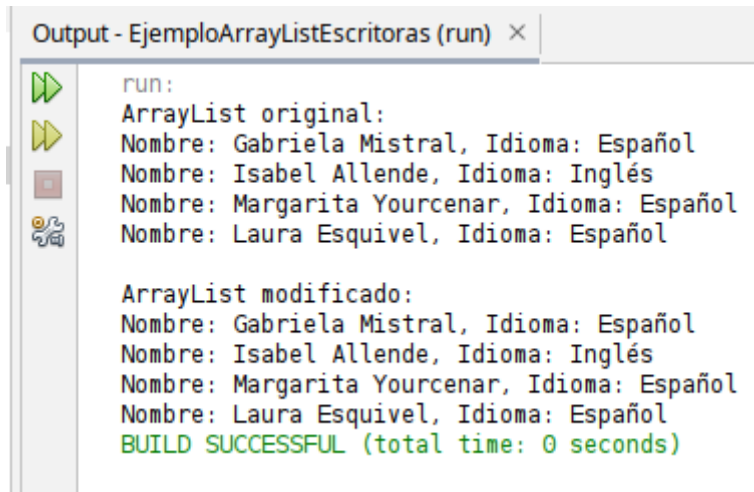
34 public class EjemploArrayList {
35     public static void main(String[] args) {
36         // Crear un ArrayList vacío
37         ArrayList<Escritora> escritoras = new ArrayList();
38
39         // Añadir cuatro objetos de clase Escritora que hablen castellano
40         escritoras.add(new Escritora("Gabriela Mistral", "Español"));
41         escritoras.add(new Escritora("Isabel Allende", "Español"));
42         escritoras.add(new Escritora("Margarita Yourcenar", "Español"));
43         escritoras.add(new Escritora("Laura Esquivel", "Español"));
44
45         // Crear un segundo ArrayList pasándole el primero
46         ArrayList<Escritora> escritorasModificadas = new ArrayList(escritoras);
47
48         // Modificar una de las escritoras en el segundo ArrayList
49         escritorasModificadas.get(1).setIdioma("Inglés");
50
51         // Imprimir ambos ArrayList
52         System.out.println("ArrayList original:");
53         for (Escritora escritora : escritoras) {
54             System.out.println("Nombre: " + escritora.getNombre() + ", Idioma: "
55                 + escritora.getIdioma());
56         }
57
58         System.out.println("\nArrayList modificado:");
59         for (Escritora escritora : escritorasModificadas) {
60             System.out.println("Nombre: " + escritora.getNombre() + ", Idioma: "
61                 + escritora.getIdioma());
62         }
63     }
64 }

```

En el ejemplo, se crea un ArrayList llamado "escritoras" que inicialmente está vacío. Luego, se añaden cuatro objetos de la clase Escritora que hablan castellano. A continuación, se crea un segundo ArrayList llamado "escritorasModificadas" pasándole el primer ArrayList.

Luego, se modifica el idioma de la segunda escritora en el segundo ArrayList. Por último, se imprimen ambos ArrayList para verificar los cambios realizados.

La salida del programa sería:



```

Output - EjemploArrayListEscritoras (run) x
run:
ArrayList original:
Nombre: Gabriela Mistral, Idioma: Español
Nombre: Isabel Allende, Idioma: Inglés
Nombre: Margarita Yourcenar, Idioma: Español
Nombre: Laura Esquivel, Idioma: Español

ArrayList modificado:
Nombre: Gabriela Mistral, Idioma: Español
Nombre: Isabel Allende, Idioma: Inglés
Nombre: Margarita Yourcenar, Idioma: Español
Nombre: Laura Esquivel, Idioma: Español
BUILD SUCCESSFUL (total time: 0 seconds)

```

LinkedList

Es muy similar en su funcionamiento externo al ArrayList: son colecciones de elementos y trabaja como una lista, pueden añadirse, eliminarse elementos ...

La diferencia está en el interior, el ArrayList trabaja como un array y cuando añade un elemento, expande el array si lo necesita. En la LinkedList, los elementos se colocan en contenedores que se enlazan unos con otros, como los vagones en un tren. Pueden añadirse elementos al final, al principio... puede ser más eficiente, dependiendo del problema que estemos solucionando.

Ver ejemplos en: https://www.w3schools.com/java/java_linkedlist.asp

Map

Es una interfaz dentro del Collections Framework , **NO EXTIENDE** a Collections, es un grupo de colecciones diferentes. Todas las clases que implementan Map tienen en común que mapean una clave con un valor. No puede existir la misma clave dos veces en un Map.

HashMap

Extiende la clase abstracta AbstractMap, que a su vez extiende a la clase raíz Objeto. AbstractMap implementa Map.

Guarda colecciones de parejas, la primera parte es la clave, que es única en la colección y la segunda el valor, que se puede repetir. Además permite valores NULL y la clave NULL.

Ver métodos: https://www.w3schools.com/java/java_hashmap.asp

Set

Es una interfaz que extiende a Collection y no puede contener elementos repetidos.

HashSet

Colecciones de objetos, donde cada uno es único. Se basa en una tabla hash, para asegurarse de que los elementos son únicos. No existe garantía de que se antenga el orden de los elementos.

Ver ejemplos: https://www.w3schools.com/java/java_hashset.asp

Otras colecciones

Existen otras implementaciones de las diferentes interfaces de colecciones, en esta

tabla pueden verse otras interesantes:

Interface	Hash Table	Resizable Array	Balanced Tree	Linked List	Hash Table + Linked List
Set	HashSet		TreeSet		LinkedHashSet
List		ArrayList		LinkedList	
Queue, Deque		ArrayDeque		LinkedList	
Map	HashMap		TreeMap		LinkedHashMap

Tabla 1: Implementaciones de colecciones

(Fuente: <https://docs.oracle.com/en/java/javase/20/docs/api/java.base/java/util/doc-files/coll-overview.html>)

Ordenando colecciones de objetos

En Java las variables y los objetos son fáciles de comparar, hasta que dejan de serlo. Si queremos comparar/ordenar objetos de la clase Persona o de la clase Pájaro, tendremos que usar otras herramientas para poder hacerlo.

Las interfaces que vamos a utilizar trabajan de manera diferente. Ambas pertenecen al package java.util

Comparable<T>

Comparable es una interfaz que permite ordenar los objetos de la clase que lo implementa, generalmente defines que una de tus clases implemente a Comparable y luego utilizas el Collection.sort() para ordenar una colección de esos objetos.

Imaginemos que tenemos una clase persona y queremos que se puedan ordenar las colecciones de personas.

Comparable tiene un método int compareTo(T o), que compara este objeto con el objeto de tipo T especificado, para saber su relación de orden. Devuelve un entero negativo, si es menor, cero si es igual o un entero positivo, si este objeto es mayor que el objeto especificado. Lanzará excepciones si el objeto es nulo o impide ser comparado.

Primero vamos a implementar Comparable en la clase Persona:

```
1 //Ejemplo generado por chatGPT el 15/06/2023
2 //Comentarios de Cristina Moreno Ruiz
3 import java.util.ArrayList;
4 import java.util.Collections;
```

```

5 import java.util.List;
6
7 //Observa cómo la clase Persona implementa la interfaz Comparable
8 class Persona implements Comparable<Persona> {
9     private String nombre;
10    private int edad;
11
12    public Persona(String nombre, int edad) {
13        this.nombre = nombre;
14        this.edad = edad;
15    }
16
17    public String getNombre() {
18        return nombre;
19    }
20
21    public int getEdad() {
22        return edad;
23    }
24    //Por lo tanto tengo que implementar mi método compareTo
25    @Override
26    public int compareTo(Persona otraPersona) {
27        // Comparamos las edades de las personas
28        // Devolvemos un valor negativo si la edad de "otraPersona" es mayor,
29        // un valor positivo si es menor, y 0 si son iguales.
30        return this.edad - otraPersona.edad;
31    }
32 }

```

Ahora en el programa principal:

```

1 public class ComparandoPersonas {
2     public static void main(String[] args) {
3         List<Persona> personas = new ArrayList<>();
4         personas.add(new Persona("Juan", 25));
5         personas.add(new Persona("Ana", 30));
6         personas.add(new Persona("Pedro", 20));
7
8         System.out.println("Lista original:");
9         for (Persona persona : personas) {
10             System.out.println(persona.getNombre() + " - " + persona.getEdad());
11         }

```

```

12    //Como la clase Persona implementa a Comparable
13    //podemos llamar al método estático sort de Collections
14    //y usará nuestro método compareTo para ordenar la colección
15    Collections.sort(personas);
16
17    System.out.println("\nLista ordenada por edad:");
18    for (Persona persona : personas) {
19        System.out.println(persona.getNombre() + " - " + persona.getEdad());
20    }
21 }
22 }

```

Los resultados por pantalla serían:

run:

Lista original:

Juan - 25

Ana - 30

Pedro - 20

Lista ordenada por edad:

Pedro - 20

Juan - 25

Ana - 30

BUILD SUCCESSFUL (total time: 0 seconds)

Comparator <T>

Podemos crear varios objetos que implementan Comparator y definir el método compareTo para cada uno de ellos, que recibe como argumentos los dos objetos a comparar. Cada clase ordenaría por un criterio, si seguimos hablando de personas, por ejemplo un comparador lo haría por edad, otro por orden alfabético de nombre, otro de ciudad...

Cualquiera de estos comparadores, se lo podemos pasar a Collection.sort(), junto con la colección a ordenar y la ordenará según el comparador recibido.

Para la misma clase Persona, pero sin necesidad de que implemente la interfaz Comparable, este sería un ejemplo de comparadores:

```

1 public class ComparadoresPersonas {
2     public static void main(String[] args) {

```

```

3 List<Persona> personas = new ArrayList<>();
4 personas.add(new Persona("Juan", 30));
5 personas.add(new Persona("María", 25));
6 personas.add(new Persona("Pedro", 35));
7
8 // Comparador por edad
9 Comparator<Persona> comparadorPorEdad = new Comparator<Persona>() {
10     @Override
11     public int compare(Persona p1, Persona p2) {
12         return Integer.compare(p1.getEdad(), p2.getEdad());
13     }
14 };
15
16 // Comparador por nombre
17 Comparator<Persona> comparadorPorNombre = new Comparator<Persona>() {
18     @Override
19     public int compare(Persona p1, Persona p2) {
20         return p1.getNombre().compareTo(p2.getNombre());
21     }
22 };
23
24 // Ordenamos la lista de personas por edad
25 Collections.sort(personas, comparadorPorEdad);
26 System.out.println("Ordenado por edad:");
27
28 for (Persona persona : personas) {
29     System.out.println(persona.getNombre() + " - " + persona.getEdad());
30 }
31
32 // Ordenamos la lista de personas por nombre
33 Collections.sort(personas, comparadorPorNombre);
34 System.out.println("\nOrdenado por nombre:");
35
36 for (Persona persona : personas) {
37     System.out.println(persona.getNombre() + " - " + persona.getEdad());
38 }
39 }
40 }

```

La salida por pantalla sería:

run:

Ordenado por edad:

María - 25

Juan - 30

Pedro - 35

Ordenado por nombre:

Juan - 30

María - 25

Pedro - 35

BUILD **SUCCESSFUL** (total time: 0 seconds)

Bibliografía

Java:

- <https://docs.oracle.com/javase/8/docs/api/>
- <https://www.w3schools.com/java>

Tipos primitivos:

- <https://docs.oracle.com/javase/tutorial/java/nutsandbolts/datatypes.html>
- https://www.w3schools.com/java/java_data_types.asp

Byte <https://docs.oracle.com/javase/8/docs/api/java/lang/Byte.html>

String <https://docs.oracle.com/javase/7/docs/api/java/lang/String.html>

String <https://docs.oracle.com/javase/7/docs/api/java/lang/String.html>

Enum https://www.w3schools.com/java/java_enums.asp

Collections Framework: <https://docs.oracle.com/javase/8/docs/technotes/guides/collections/index.html>

LinkedList:

- <https://docs.oracle.com/javase/8/docs/api/java/util/LinkedList.html>
- https://www.w3schools.com/java/java_linkedlist.asp

HashMap:

- https://www.w3schools.com/java/java_hashmap.asp
- <https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html>

HashSet:

- https://www.w3schools.com/java/java_hashset.asp
- <https://docs.oracle.com/en/java/javase/12/docs/api/java.base/java/util/HashSet.html>

Comparable:

<https://docs.oracle.com/en/java/javase/20/docs/api/java.base/java/lang/Comparable.html>

Comparator:

<https://docs.oracle.com/en/java/javase/20/docs/api/java.base/java/util/Comparator.html>

<https://www.baeldung.com/java-comparator-comparable>

Traducción e interpretación: Cristina Moreno Ruiz