Colecciones
Ordenación y conjuntos

La clase HashSet

Es una colección en la que los elementos no se guardan ordenados (internamente se ordenan para mantener el rendimiento en la gestión de la colección).

No admite repetición de objetos. El método add() se encarga automáticamente de la comprobación.

Para comprobar si un objeto está en el HashSet, comprueba el hash de ese objeto.

Si queremos modificar el criterio por el que se comprueba si un objeto ya está en la colección, debemos sobreescribir el método hashCode() de Object, según nos convenga.

La clase TreeSet

Es una colección en la que los elementos se guardan ordenados según el “orden natural” de la clase correspondiente a los objetos.

No admite repetición de objetos. El método add() se encarga automáticamente de la comprobación.

Para que se pueda comprobar si un objeto está en el TreeSet, la clase de los objetos debe implementar la interfaz Comparable, que obliga a definir el método compareTo, que es el que se utiliza (si devuelve 0 los objetos son iguales).

Además, el TreeSet está ordenado según ese orden natural definido en la clase según compareTo (si nos devuelve un negativo el objeto es menor. Si nos devuelve un positivo, es mayor).

Recuerda que en el caso de que los objetos sean, por ejemplo, del tipo String, ya existe el orden natural porque la clase String implementa la interfaz Comparable.

Orden en Listas

La clase LinkedList y ArrayList difieren en su estructura interna, lo que hace que el rendimiento de algunas operaciones sea diferente en una y otra.

En cuanto al orden, la forma de trabajar es igual en las dos.

Para poder ordenarlas recurrimos a la clase Collections, que contiene métodos estáticos (podemos invocarlos sin crear instancias, simplemente llamando al método de clase). Dispone del método sort, que ORDENA la lista que le pasamos como argumento.

Está sobrecargado, por lo que si solo le pasamos como argumento la lista a ordenar, la ordenará según el “orden natural” de la clase (es decir, como hemos visto antes según el método compareTo).

También tenemos opción de definir órdenes alternativos, creando clases que implementen la interfaz Comparator, que exige la creación de un método equals y un compare (en realidad equals estará presente como herencia de la clase object, pero debería sobreescribirse para asegurar la coherencia con el método compare cuando dos objetos se definan como iguales).

Para aplicar el orden definido en una clase que implemente Comparator, debemos invocar al método sort sobrecargado en el que pasamos como argumento además de la lista a ordenar, una instancia de la clase “comparadora”.

Nota: También se puede utilizar el método sort(Comparator c) en las clases que implementan la interfaz List, como ArrayList