Daniel Salomón Marmolejo Contreras 5D

Introducción

Las colecciones representan grupos de objetos, denominados elementos. Podemos encontrar diversos tipos de colecciones, según si sus elementos están ordenados, o si permitimos repetición de elementos o no.

Es el tipo más genérico en cuanto a que se refiere a cualquier tipo que contenga un grupo de elementos. Viene definido por la interfaz Collection, de la cual heredará cada subtipo específico. En esta interfaz encontramos una serie de métodos que nos servirán para acceder a los elementos de cualquier colección de datos, sea del tipo que sea. Estos métodos generales son:

boolean add(Object o)

Añade un elemento (objeto) a la colección. Nos devuelve true si tras añadir el elemento la colección ha cambiado, es decir, el elemento se ha añadido correctamente, o false en caso contrario.

void clear()

Elimina todos los elementos de la colección.

boolean contains(Object o)

Indica si la colección contiene el elemento (objeto) indicado.

boolean isEmpty()

Indica si la colección está vacía (no tiene ningún elemento).

Iterator iterator()

Proporciona un iterador para acceder a los elementos de la colección.

boolean remove(Object o)

Elimina un determinado elemento (objeto) de la colección, devolviendo true si dicho elemento estaba contenido en la colección, y false en caso contrario.

int size()

Nos devuelve el número de elementos que contiene la colección.

Object [] toArray()

Nos devuelve la colección de elementos como un array de objetos. Si sabemos de antemano que los objetos de la colección son todos de un determinado tipo (como por ejemplo de tipo String) podremos obtenerlos en un array del tipo adecuado, en lugar de usar un array de objetos genéricos.

Desarrollo

Listas de elementos

Este tipo de colección se refiere a listas en las que los elementos de la colección tienen un orden, existe una secuencia de elementos. En ellas cada elemento estará en una determinada posición (índice) de la lista.

Las listas vienen definidas en la interfaz List, que además de los métodos generales de las colecciones, nos ofrece los siguientes para trabajar con los índices:

void add(int indice, Object obj)

Inserta un elemento (objeto) en la posición de la lista dada por el índice indicado.

Object get(int indice)

Obtiene el elemento (objeto) de la posición de la lista dada por el índice indicado.

int indexOf(Object obj)

Nos dice cual es el índice de dicho elemento (objeto) dentro de la lista. Nos devuelve -1 si el objeto no se encuentra en la lista.

Object remove(int indice)

Elimina el elemento que se encuentre en la posición de la lista indicada mediante dicho índice, devolviéndonos el objeto eliminado.

Object set(int indice, Object obj)

Establece el elemento de la lista en la posición dada por el índice al objeto indicado, sobrescribiendo el objeto que hubiera anteriormente en dicha posición. Nos devolverá el elemento que había previamente en dicha posición.

Podemos encontrar diferentes implementaciones de listas de elementos en Java:

ArrayList: Representa una lista ordenada pero no clasificada. La iteración o recorrido por esta colección es muy rápida.

Vector: Igual que ArrayList, pero sus métodos son Synchronized (Ya lo explicaremos más adelante)

LinkedList: Representa una lista ordenada y clasificada. La iteración o recorrido por esta colección es más lenta que por ArrayList o Vector.

Conjuntos

Los conjuntos son grupos de elementos en los que no encontramos ningún elemento repetido. Consideramos que un elemento está repetido si tenemos dos objetos o1 y o2 iguales, comparandolos mediante el operador o1.equals(o2). De esta forma, si el objeto a insertar en el conjunto estuviese repetido, no nos dejará insertarlo. Recordemos que el método add devolvía un valor booleano, que servirá para este caso, devolviendonos true si el elemento a añadir no estaba en el conjunto y ha sido añadido, o false si el elemento ya se encontraba dentro del conjunto. Un conjunto podrá contener a lo sumo un elemento null.

Los conjuntos se definen en la interfaz Set, a partir de la cuál se construyen diferentes implementaciones:

HashSet

Los objetos se almacenan en una tabla de dispersión (hash). El coste de las operaciones básicas (inserción, borrado, búsqueda) se realizan en tiempo constante siempre que los elementos se hayan dispersado de forma adecuada. La iteración a través de sus elementos es más costosa, ya que necesitará recorrer todas las entradas de la tabla de dispersión, lo que hará que el coste esté en función tanto del número de elementos insertados en el conjunto como del número de entradas de la tabla. El orden de iteración puede diferir del orden en el que se insertaron los elementos.

LinkedHashSet

Es similar a la anterior pero la tabla de dispersión es doblemente enlazada. Los elementos que se inserten tendrán enlaces entre ellos. Por lo tanto, las operaciones básicas seguirán teniendo coste constante, con la carga adicional que supone tener que gestionar los enlaces. Sin embargo habrá una mejora en la iteración, ya que al establecerse enlaces entre los elementos no tendremos que recorrer todas las entradas de la tabla, el coste sólo estará en función del número de elementos insertados. En este caso, al haber enlaces entre los elementos, estos enlaces definirán el orden en el que se insertaron en el conjunto, por lo que el orden de iteración será el mismo orden en el que se insertaron.

TreeSet

Utiliza un árbol para el almacenamiento de los elementos. Por lo tanto, el coste para realizar las operaciones básicas será logarítmico con el número de elementos que tenga el conjunto O(log n).

Mapas

Aunque muchas veces se hable de los mapas como una colección, en realidad no lo son, ya que no heredan de la interfaz Collection.

Los mapas se definen en la interfaz Map. Un mapa es un objeto que relaciona una clave (key) con un valor. Contendrá un conjunto de claves, y a cada clave se le

asociará un determinado valor. En versiones anteriores este mapeado entre claves y valores lo hacía la clase Dictionary, que ha quedado obsoleta. Tanto la clave como el valor puede ser cualquier objeto.

Los métodos básicos para trabajar con estos elementos son los siguientes:

Object get(Object clave)

Nos devuelve el valor asociado a la clave indicada

Object put(Object clave, Object valor)

Inserta una nueva clave con el valor especificado. Nos devuelve el valor que tenía antes dicha clave, o null si la clave no estaba en la tabla todavía.

Object remove(Object clave)

Elimina una clave, devolviendonos el valor que tenía dicha clave.

Set keySet()

Nos devuelve el conjunto de claves registradas

int size()

Nos devuelve el número de parejas (clave,valor) registradas.

Encontramos distintas implementaciones de los mapas:

HashMap

Utiliza una tabla de dispersión para almacenar la información del mapa. Las operaciones básicas (get y put) se harán en tiempo constante siempre que se dispersen adecuadamente los elementos. Es coste de la iteración dependerá del número de entradas de la tabla y del número de elementos del mapa. No se garantiza que se respete el orden de las claves.

TreeMap

Utiliza un árbol rojo-negro para implementar el mapa. El coste de las operaciones básicas será logarítmico con el número de elementos del mapa O(log n). En este caso los elementos se encontrarán ordenados por orden ascendente de clave.

Hashtable

Es una implementación similar a HashMap, pero con alguna diferencia. Mientras las anteriores implementaciones no están sincronizadas, este si que lo está. Además, en esta implementación, al contrario que las anteriores, no se permitirán claves nulas (null). Este objeto extiende la obsolea clase Dictionary, ya que viene de versiones más antiguas de JDK. Ofrece otros métodos además de los anteriores, como por ejemplo el siguiente:

Enumeration keys()

Este método nos devolverá una enumeración de todas las claves registradas en la tabla.

Conclusión

Java proporciona una serie de estructuras muy variadas para almacenar datos. Estas estructuras, ofrecen diversas funcionalidades: ordenación de elementos, mejora de rendimiento, rango de operaciones… Es importante conocer cada una de ellas para saber cuál es la mejor situación para utilizarlas. Un buen uso de estas estructuras mejorará el rendimiento de nuestra aplicación.