Tema VIII: COLECCIONES DE DATOS

"Caminar sobre el agua y desarrollar software a partir de unas especificaciones es fácil,
si ambas están congeladas."

Cuando utilizamos un array estamos obligados a solicitar "espacio" para un número determinado de elementos y todos del mismo tipo. Quizás en tiempo de ejecución no necesitemos todo la memoria solicitada o por el contrario necesitemos más.
Java, al igual que el resto de lenguajes orientado a objetos, dispone de clases que nos permiten almacenar igual que un array pero que se incrementan automáticamente cuándo ocupamos todo el espacio inicial. Dichas clases se llaman colecciones. Se encuentran en el paquete util

COLECCIONES

- Collection<E>: Un grupo de elementos individuales, frecuentemente con alguna regla aplicada a ellos.
- List<E>: Elementos en una secuencia particular que mantienen un orden y permite duplicados. La lista puede ser recorrida en ambas direcciones. Hay 3 tipos

1. ArrayList<E>: Su ventaja es que el acceso a un elemento en particular es ínfimo. Su desventaja es que para eliminar un elemento, se ha de mover toda la lista para eliminar ese “hueco”.

2. Vector<E>: Es igual que ArrayList, pero sincronizado. Es decir, que si usamos varios hilos, no tendremos de qué preocuparnos hasta cierto punto.

3. LinkedList<E>: En esta, los elementos están conectados con el anterior y el posterior. La ventaja es que es fácil mover/eliminar elementos de la lista, simplemente moviendo/eliminando sus referencias hacia otros elementos. La desventaja es que para usar el elemento N de la lista, debemos realizar N movimientos a través de la lista.

- **Map<K,V> y HashMap<K,V>**: Un grupo de pares objeto clave-valor, que no permite duplicados en sus claves. Es quizás el más sencillo, y no utiliza la interfaz Collection. Los principales métodos son: put(), get(), remove().

- **Set<E> y HashSet<E>**: No puede haber duplicados. Cada elemento debe ser único, por lo que si existe uno duplicado, no se agrega. Por regla general, cuando se redefine equals(), se debe redefinir hashCode(). Es necesario redefinir hashCode() cuando la clase definida será colocada en un HashSet. Los métodos add(o) y addAll(o) devuelven false si o ya estaba en el conjunto.

- **Queue<E>**: Colección ordenada con extracción por el principio e inserción por el principio (LIFO – Last Input, First Output) o por el final (FIFO – First Input, First Output). Se permiten elementos duplicados. No da excepciones cuando la cola está vacía/llena, hay métodos para interrogar, que devuelven null. Los métodos put()/take() se bloquean hasta que hay espacio en la cola/haya elementos.

VECTORES

Un Vector es un objeto que sirve para almacenar otros objetos. Es parecido a un array pero se diferencia en:

- Un vector crece y decrece automáticamente dependiendo de los elementos que tenga almacenados. No es necesario darle el nº de elementos al construirlo.
- Un array forma parte del lenguaje Java, un Vector es una clase del paquete util.
- Un array puede contener tipos básicos u objetos, un Vector sólo contiene objetos, estos objetos son siempre de la clase Object.

Crear un vector

- Vector nombre=new Vector();

Crea un vector con capacidad inicial de 10 objetos y cuando se llena incrementa su capacidad en el doble
Ej.

```
Vector listaClase=new Vector ();
```

- Vector nombre=new Vector(nº elementos);

Crea un vector con capacidad para inicialmente ese número de elementos y cuando se llena incrementa su capacidad en el doble.

Ej.

```
Vector listaCoches=new Vector (6);
```

- Vector nombre=new Vector(3,4);

Vector con capacidad para 3 elementos y que cuando se llena se incrementa de 4 en 4.

En todos los casos anteriores el vector se construye sin indicar el tipo de elementos que contendrá. Se asume que son de tipo Object.

Se podrán almacenar elementos de cualquier tipo, pero para obtenerlos desde el vector tendremos que hacer un casting.

Podemos "parametrizar". Construimos el vector indicando el tipo de elementos que contendrá. De esta forma, evitaremos las operaciones de casting.

Vector<tipo objetos> nombre=new Vector<tipo objetos>();

Tamaño

int size(): Retorna el número de elementos que tiene almacenados el vector.

Añadir un elemento

- boolean add(Object obj)
- boolean add(Tipo declarado obj) si hemos parametrizado

Añade un elemento al final del Vector.

- void add(int index, Object obj)

Añade un elemento en la posición index (entre 0 y size()) y desplaza todos los elementos a la derecha.

Obtener un elemento

- Object get(int index)
- (Tipo declarado)get(int index)

Devuelve el objeto almacenado en la posición index del vector, el índice debe estar comprendido entre 0 y size()-1. No saca el elemento.

Modificar un elemento:

- void set(int index,Object obj)

Mueve el valor de obj a la posición index del vector. La posición debe estar comprendida entre 0 y size()-1

Borrar un elemento:

- Object remove(int index)

Borra un elemento en la posición index (de 0 a size()-1) y mueve todos los elementos a la izda. Devuelve el elemento borrado.

- boolean remove(Object obj)

Borra el objeto obj en el vector. Si lo encuentra retorna true, si no lo encuentra retorna false. Utiliza el método equals para comparar los objetos

Buscar un elemento:

- int indexOf (Object obj)

Usando el método equals nos dice si ya existe un elemento en un vector (también sirve para arrays). Para ello tenemos que redefinir el método equals para los objetos almacenados en el vector. Retorna –1 si no lo encuentra y si lo encuentra el índice donde lo ha encontrado.

NOTAS

Puedo meter objetos de cualquier tipo en un vector, y utilizar polimorfismo igual que en los arrays.

Es mejor usar vectores cuando los objetos a almacenar son pequeños. Si son grandes el mantenimiento es bastante más costoso que con un array.

Utilizando la documentación de java pueden verse otros métodos que tiene esta clase.

ARRAYLIST Y LINKEDLIST

Funcionan igual que los vectores, y tienen los mismos métodos que los vectores y alguno más.

Un ArrayList, igual que un Vector, internamente almacena la información en un array, mientras que un LinkedList lo hace en una lista doblemente enlazada.

Por este motivo, en un ArrayList el acceso a un elemento es muy rápido, mientras que el borrado y la inserción son costosos. Justo alrevés que en un LinkedList, dónde insertar y borrar es sencillo pero acceder a un elemento es costoso porque tiene que recorrerse todos los anteriores.

Por lo tanto lo mejor es usar ArrayList para almacenar y acceder a datos y usar LinkedList cuando voy a hacer muchas altas y modificaciones.

La clase LinkedList tiene varios métodos para hacer estas operaciones más eficientes:

- addFirst(objeto) : Añade un elemento al principio
- addLast(objeto) : Añade un elemento al final
- removeFirst(): Borra el primer elemento
- removeLast(): Borra el último elemento

Ejemplo uso de ArrayList:

```
public class Main {
  public static void main(String[] args) {
    ArrayList<String> cars = new ArrayList<String>();
    cars.add("Volvo");
    cars.add("BMW");
    cars.add("Ford");
    cars.add("Mazda");
    for (int i = 0; i < cars.size(); i++) {
      System.out.println(cars.get(i));
    }
  }
}
```

Ejemplo uso de LinkedList:

```
public class Main {
  public static void main(String[] args) {
    LinkedList<String> cars = new LinkedList<String>();
    cars.add("Volvo");
    cars.add("BMW");
    cars.add("Ford");
    cars.add("Mazda");
    for (int i = 0; i < cars.size(); i++) {
      System.out.println(cars.get(i));
    }
  }
}
```

Ejemplo uso de ArrayList con bucle for-each (también se puede usar con LinkedList):

```
public class Main {
  public static void main(String[] args) {
    ArrayList<String> cars = new ArrayList<String>();
    cars.add("Volvo");
    cars.add("BMW");
    cars.add("Ford");
    cars.add("Mazda");
    for (String i : cars) {
      System.out.println(i);
    }
  }
}
```

HASHMAP

Almacena los datos en parejas: clave,valor.
Se usa un objeto para la clave y otro para el valor.

Ejemplo de HashMap, que almacena parejas nombre, edad. La clave es el nombre:

```
HashMap<String, Integer> people = new HashMap<String, Integer>();
```

Métodos más importantes:

- put(clave,valor) : Introduce una clave con su valor asociado. Si existe esa clave sobreescribe el valor.
- get(clave): Accede al valor asociado a esa clave, si no existe devuelve null.
- containsKey(clave): Devuelve true o false dependiendo si existe un dato con esa clave o no.
- Remove(clave): Borra el dato con dicha clave.

Se recorre con for_each

Ejemplo HashMap:

```
import java.util.HashMap;


public class Main {
  public static void main(String[] args) {

    // Create a HashMap object called people
    HashMap<String, Integer> people = new HashMap<String, Integer>();



    // Add keys and values (Name, Age)
    people.put("John", 32);
    people.put("Steve", 30);
```

```
    people.put("Angie", 33);

    for (String i : people.keySet()) {
      System.out.println("key: " + i + " value: " + people.get(i));
    }
  }
}
```

HASHSET

Es una colección de datos dónde cada dato es único.

Ejemplo:

```
HashSet<String> cars = new HashSet<String>();
```

Métodos más importantes:

- add(obj): añade un objeto si no existe.
- contains(obj): indica si el objeto está en el conjunto.
- size(): Número de elementos.
- Remove(obj): borra un objeto

Se recorre con for_each.

Ejemplo HasSet:

```
import java.util.HashSet;

public class Main {
  public static void main(String[] args) {

    // Create a HashSet object called numbers
    HashSet<Integer> numbers = new HashSet<Integer>();

    // Add values to the set
    numbers.add(4);
```

```
    numbers.add(7);
    numbers.add(8);

    // Show which numbers between 1 and 10 are in the set
    for(int i = 1; i <= 10; i++) {
      if(numbers.contains(i)) {
        System.out.println(i + " was found in the set.");
      } else {
        System.out.println(i + " was not found in the set.");
      }
    }
  }
}
```