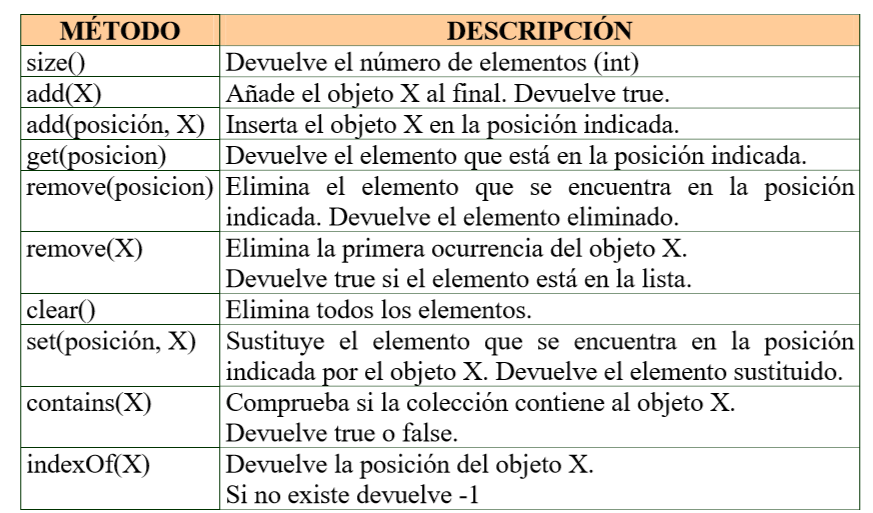
La clase **ArrayList**



La clase **ArrayList**

Características importantes de la clase ArrayList son:

* Es capaz de aumentar su capacidad interna tanto como se requiera.
* Cuando se agregan más elementos, simplemente crea el espacio necesario para ellos.
* Mantiene su propia cuenta privada de la cantidad de elementos almacenados, que se obtiene mediante size.
* Mantiene el orden de los elementos que se agregan.
* El método add almacena cada nuevo elemento al final de la lista.
* Posteriormente se pueden recuperar en el mismo orden.

Tipos primitivos y la clase **ArrayList**

Los elementos de los **ArrayList** son objetos (**Object**).

Los enteros, por ejemplo, no son objetos.

Por tanto, un **int** no se puede insertar en un **ArrayList**.

¿Qué hacemos entonces? Utilizar la clase envoltorio **Integer**.

**import** java.util.ArrayList;

**public** **class** DemoEnvoltorio {

**public** **static** **void** main(String[] args) {

ArrayList<Integer> lista = **new** ArrayList<Integer>();

lista.add(**new** Integer(42));

Integer n;

n = lista.get(0);

**int** miEntero = n;

System.***out***.println(n);

}

}

Otras clases envoltorios son **Byte**, **Float**, **Double**, **Long** y **Short**.

La interfaz **Iterator**

Los iteradores permiten recorrer colecciones sin preocuparse de la implementación subyacente.

**hasNext()**: ¿hay un elemento siguiente?

**next()**: devuelve el siguiente elemento de la colección.

**remove()**: elimina el último elemento devuelto por el iterador.

**public void imprime(Collection col){ Iterator it = col.iterator(); while(it.hasNext())**

**System.out.println(it.next());**

**}**

Iteradores sobre distintos tipos de colecciones

**import** java.util.\*;

**public** **class** PruebaColecciones{

**public** **static** **void** prueba(Collection c) {

String[] lista = {"uno", "dos" , "tres", "cuatro", "tres"};

**for**(**int** i = 0; i < lista.length; i++ )

c.add( lista[i] );

Iterator it = c.iterator();

**while**(it.hasNext())

System.***out***.println( it.next() );

System.***out***.println( "-------------------------");

}

**public** **static** **void** main(String args[]) { Collection c;

c = **new** ArrayList(); *prueba*(c);

c = **new** LinkedList(); *prueba*(c);

c = **new** HashSet(); *prueba*(c);

c = **new** TreeSet(); *prueba*(c);

}

}

Más formas para recorrer colecciones

Bucle for each  Se incorpora en la versión 5 de Java. Esta estructura nos permite recorrer una Colección o un array de elementos de una forma sencilla. Evita el uso de Iteradores o de un bucle for normal

// Declaro un ArrayList ArrayList<String> pelis = new ArrayList<String>();

// Estructura for each

for (String nombre: pelis)

System.out.println(nombre);

El resultado es más legible. Sin embargo en ocasiones los iteradores aportan cosas interesantes que los bucles for each no pueden abordar. Vamos a borrar todos los objetos de una lista con un nombre concreto. La operación parece tan sencilla como hacer lo siguiente:

for (String nombre:pelis)

{ if (nombre.equals("Casablanca"))

{ pelis.remove("Casablanca"); } }

El código no funciona y lanza una excepción.

El interface Iterator dispone de un método adicional que permite eliminar objetos de una lista mientras la recorremos. El método remove.

La clase **LinkedList**

Esta clase organiza los elementos de una colección como una lista doblemente enlazada. Las operaciones de inserción y borrado en posiciones intermedias son muy eficientes. Por el contrario, el acceso a un elemento por índice es ineficiente. Más rápida que ArrayList añadiendo elementos al principio y eliminando elementos en general.

La debemos utilzar en lugar de ArrayList si se realizan más operaciones de inserción, en posición 0, o de eliminación que de lectura. La diferencia de rendimiento es enorme. Las búsquedas o las inserciones en posiciones intermedias de una lista con gran número de elementos es más rápida en el caso de ArrayList. Esto es debido a que ésta accede a la posición de manera directa.

La clase dispone de un constructor sin argumentos que crea una lista vacía, y otro constructor que crea la lista con los elementos de otra colección.

public LinkedList(); public LinkedList(Collection c);

Implementa la interfaz Cloneable.

Las operaciones generales de las listas y métodos específicos que operan sobre el primer y último elemento son: public Object getFirst() public Object getLast() public void addFirst(Object ob) public void addLast(Object ob) public Object removeFirst() public Object removeFirst()

La clase **LinkedList**

Se puede usar para crear una estructura de datos Cola o Pila. Añadir datos a la Cola se hace con addLast(). Eliminar con removeFirst(). Obtener el primer elemento con getFirst().

En cuanto a ArrayList y LinkedList vemos que las inserciones en posiciones iniciales o finales son por regla general más rápidas con LinkedList. Igualmente pasaría con las eliminaciones o borrados. Aunque observamos que las búsquedas o las inserciones en posiciones intermedias de una lista con gran número de elementos es más rápida en el caso de ArrayList. Esto es debido a que ésta accede a la posición de manera directa.

Para finalizar esta comparación vemos que el método add por defecto sin posición es más rápido que insertar en posiciones iniciales o intermedias en ambas implementaciones.