ARRAYLIST

¿QUÉ ES UN ARRAYLIST?

Un **ArrayList** es una estructura que permite almacenar datos de forma similar a como lo realizaría un Array, con la diferencia que se trata de un **almacenamiento dinámico**; es decir, no tiene un tamaño establecido pudiendo añadir tantos elementos como queramos y borrarlos si no nos hacen falta. El espacio se va adaptando de forma dinámica y automáticamente según se vaya necesitando.

DECLARACIÓN Y PRINCIPALES MÉTODOS

De forma general, un ArrayList puede contener objetos de cualquier tipo y se declara como:

ArrayList<tipo> nombreArray = new ArrayList<>();

tipo debe ser una clase (no se pueden usar tipos primitivos).

Los siguientes métodos nos permiten realizar operaciones comunes tales como añadir, eliminar, buscar y modificar datos en un ArrayList.

MÉTODO	DESCRIPCIÓN
size()	Devuelve el número de elementos (int)
add(X)	Añade el objeto X al final.
add(posición, X)	Inserta el objeto X en la posición indicada.
get(posicion)	Devuelve el elemento que está en la posición indicada.
remove(posicion)	Elimina el elemento que se encuentra en la posición indicada.
remove(objeto)	Elimina el objeto referenciado que se le pasa como parametro
clear()	Elimina todos los elementos.
set(posición, X)	Sustituye el elemento que se encuentra en la posición indicada por el objeto X.

Cuando queramos usar ArrayList con tipos simples como char, int, boolean, double usaremos **clases envolventes**. Esto es así porque ArrayList solo puede almacenar clases/objetos.

- **int** le corresponde **Integer**.
- **double** le corresponde **Double**.
- **boolean** le corresponde **Boolean**.
- **char** le corresponde **Character**.

A continuación, se muestra el uso de alguno de los métodos disponibles en ArrayList

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

public class EjemploArrayList {

    public static void main(String[] args) {
        Scanner teclado=new Scanner(System.in);
        ArrayList<String> lista_nombres=new ArrayList<>();
        String dato;

        for (int i = 1; i <= 5; i++) {
            System.out.println("Dime un nombre");
            dato=teclado.next();
            lista_nombres.add(dato);
        }

        System.out.println("Cantidad de nombre: "+lista_nombres.size());
        System.out.println("Tercer nombre: "+lista_nombres.get(2));
        //Borrar primera posicion
        lista_nombres.remove(0);
        //Sustituye la posicion 3 por Juan"
        lista_nombres.set(3, "Juan");
```

Para recorrerlos completamente podemos usar un bucle for.

```
for(int i=0;i<lista_nombres.size();i++){
    System.out.println(lista_nombres.get(i));
}
```

El método get si vamos a recorrer el ArrayList completamente no es eficiente y desaconsejado debido a que las posiciones no son fijas ya que el objetivo de estas colecciones es añadir/borrar datos constantemente y por ello no está optimizado para acceder por posición (para ello usar un array normal).
Por ello se proporciona una variante de bucle for **(tipo for each)** que no es necesario usar los métodos y directamente nos coloca uno a uno cada elemento del ArrayList. Por cada elemento de la **lista_nombre** el bucle da una vuelta y coloca el elemento en orden en la variable **nombre**. Si no necesitamos saber la posición, de esta forma es más compacto.

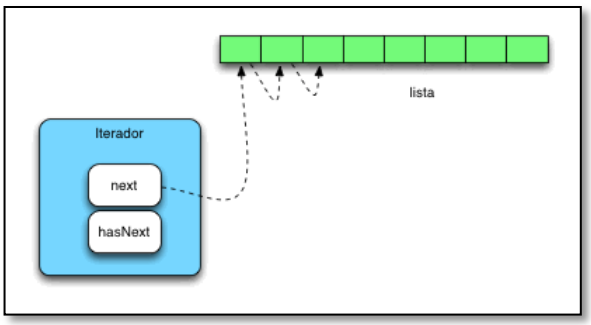
```
for (String nombre : lista_nombres) {
    System.out.println(nombre);
}
```

El bucle anterior recorre siempre completamente los datos del ArrayList y además si queremos eliminar un dato en concreto que estemos buscando obtenemos lo siguiente:

```
for(Cerveza objeto:lista){
    if(objeto.getPrecio()>10){
        lista.remove(objeto);
    }
}
```

```
run:
Exception in thread "main" java.util.ConcurrentModificationException
        at java.util.ArrayList$Itr.checkForComodification(ArrayList.java:909)
        at java.util.ArrayList$Itr.next(ArrayList.java:859)
        at ejercicio6.Ejercicio6.main(Ejercicio6.java:33)
C:\Users\Daniel\AppData\Local\NetBeans\Cache\8.2\executor-snippets\run.xml:53: Java returned: 1
BUILD FAILED (total time: 1 second)
```

Para solucionar estas problemáticas tenemos un concepto de iterador



Un iterador es una estructura que nos controla cuando se llega al final del array sin necesidad de controlar una variable índice (la i del for de toda la vida) y nos proporciona uno a uno cada uno de los objetos que forman parte de la colección. Es una forma de recorrer un ArrayList más eficiente que **get(posicion)** y la usaremos en su lugar.
En un principio el ArrayList tiene un método **iterator()** que devuelve objeto iterador que en esencia solo tiene el comportamiento propio del iterador que actúa sobre ese ArrayList. (Mirar la figura anterior). Para ello se usa una definición que es la clase Iterator que al igual que ArrayList es genérica usando <TipoDedatos>. Por ejemplo, para recorrer una ArrayList de cervezas podemos usar este código:

```
ArrayList<Cerveza> lista=new ArrayList<>();
//...Rellenar datos

Iterator<Cerveza> it=lista.iterator();
Cerveza objeto;
while(it.hasNext()){//True si quedan objetos
    objeto=it.next();//nos da el objeto y avanza al siguiente
    System.out.println("Nombre:"+objeto.getNombre());
}
```

- El método **hasNext** devuelve un booleano que es true mientras haya un objeto por recorrer, o sea cuando llegue al último e invoquemos a **hasNext** devolverá false.

- El método **next** nos devuelve el siguiente objeto de la lista que toca y avanza al siguiente (Mirar la figura anterior). Si **hasNext** devuelve false e intentamos hacer **next** ocurrirá una excepción y el programa parará.
  Para resolver la situación anterior y borrar un conjunto de cervezas:

```
Iterator<Cerveza> it=lista.iterator();
Cerveza objeto;
while(it.hasNext()){
    objeto=it.next();
    if(objeto.getPrecio()>10){
        it.remove();//Borra el elemento en el que se encuentra
    }
}
```

Si queremos buscar un nombre de cerveza en concreto y poder usar un while que es la forma más eficiente:

```
Cerveza res = null, cerveza;
Iterator<Cerveza> it = this.almacen.iterator();

while (it.hasNext() && res == null) {
    cerveza = it.next();
    if (cerveza.getNombre().equalsIgnoreCase(nombre)) {
        res = cerveza;
    }
}
```

En este caso no usamos la posición sino una referencia al objeto que queremos buscar, siendo **null** al principio porque indica que no lo hemos encontrado.
El concepto de iterador es importante porque muchas estructuras son colecciones, pero no tienen por qué estar organizadas posicionalmente como veremos los ficheros o las bases de datos, etc. El iterador proporciona independencia de planteamiento independientemente del lenguaje y de la colección.
El ArrayList se puede ordenar con el método sort. Este método necesita que le indiquemos bajo qué criterio ordenarlo en base a los objetos que contiene. Por ejemplo, si contiene String podemos usar el método compareTo de la clase String

```
//orden ascendente
palabras.sort((a, b) -> a.compareToIgnoreCase(b));
```

En caso de ser la cervecería y queremos mostrarlos por nombre debemos seleccionar su nombre y aplicar el método compareTo.

```
//Orden ascendente
cervezas.sort((a, b) -> a.getNombre().compareToIgnoreCase(b.getNombre()));
```

Este método devuelve el orden relativo de dos String mediante números:

- Si el número es negativo significa que a está en orden con b.
- Si es cero son iguales en orden.
- Si el número es positivo significa que a esta en desorden con b.

Si queremos el orden inverso es tan sencillo como hacer la comparación de y b al revés

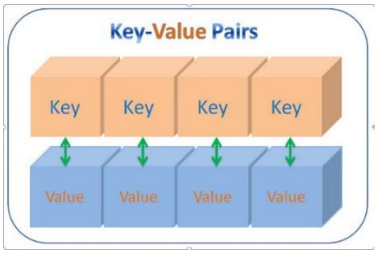
```
//Orden descendente
cervezas.sort((a, b) -> a.getNombre().compareToIgnoreCase(b.getNombre()));
```

Si queremos comparar por un valor numérico debemos usar los métodos compareTo de la clase Integer o de la clase Double según el tipo de dato. Para ordenar por precio descendente sería:

```
cervezas.sort((a, b) -> Double.compare(b.getPrecio(), a.getPrecio()));
```

HASHMAP

Dentro de las colecciones existe un tipo especial llamadas HashMap o también llamados diccionarios. Son colecciones que no se organizan posicionalmente sino en forma clave->valor (key->value). Se les llama diccionarios justamente por parecerse a un diccionario cuyo contenido está asociado a cómo encontrar el contenido, o sea, si quiero buscar la definición de mesa busco la palabra mesa, no voy buscando la posición en la que está.



Como ejemplo podríamos pensar en que una clave puede ser un número de pasaporte de una persona y como valor el objeto Persona que contiene toda la información y métodos de una persona en concreto. Aunque la clave lo más frecuente es que sea algo conciso (como un número) también puede ser otra cosa. Por ejemplo, el objeto clave podría ser una imagen, al que le correspondiera por ejemplo otra imagen valor. Por ejemplo, podríamos tener un mapa de conversión de imágenes, que, dada una imagen clave, le correspondiera una imagen valor que fuera la misma imagen escalada a un determinado tamaño. Esto no es lo más usual, pero sería posible.

Ejemplo de uso como un diccionario de español a inglés:

```
HashMap<String,String> diccionario=new HashMap<>();
diccionario.put("gato", "cat");
diccionario.put("perro", "dog");
diccionario.put("España", "Spain");
diccionario.put("pelo","hair");
```

Este tipo de dato hoy día se encuentra de una manera u otro en la mayoría de las aplicaciones en todo el mundo ya que tiene posibilidades infinitas. En PHP se llaman arrays asociativos, también se usan para intercambio de información como los objetos JSON usados en JavaScript y otros lenguajes. Por tanto, es de gran utilidad comprender su funcionamiento.

```
System.out.print("Que palabra quieres traducir al inglés:");
español=teclado.next();
System.out.println(diccionario.get(español));
```

Como acabamos de ver podemos encontrar fácilmente un objeto a través de clave, pero además un HashMap aporta las siguientes funcionalidades respecto a su contenido:
Puede que una clave no exista y en ese caso lo anterior devolvería null. Para comprobar si una clave existe tenemos el método containsKey

```
System.out.print("Que palabra quieres traducir al inglés:");
español=teclado.next();

if(diccionario.containsKey(español)){
    System.out.println(diccionario.get(español));
}else{
    System.out.println("No existe traduccion");
}
```

También podemos comprobar si un valor concreto está asociado a una clave, para ello usamos el método containsValue:

```
System.out.print("Palabra en ingles:");
ingles=teclado.next();
if(diccionario.containsValue(ingles)){
    System.out.println("Palabra correcta en inglés");
}else{
    System.out.println("No existe esa palabra en inglés");
}
```

Pero no podemos saber que clave le correspondería (correspondencia en sentido contrario), en este caso si es muy necesario tener una correspondencia bidireccional podríamos un HashMap en sentido inverso, depende de las necesidades de nuestra aplicación.
Los HashMap como colecciones que son también pueden ser iterados de la misma manera que un ArrayList, solo que en este caso existen dos colecciones, las claves y los valores. Los podemos tratar como tal individualmente de las siguientes maneras:

- Recorrer las claves:

```
for (String pal_esp : diccionario.keySet()) {
    System.out.println(pal_esp);
}
```

- Recorrer los valores:

```
for (String pal_ing : diccionario.values()) {
    System.out.println(pal_ing);
}
```

- Recorrer los valores a través de recorrer las claves (no está recomendado):

Los HashMap no están organizados de manera continua en memoria por lo que realmente estaríamos haciendo algo tremendamente ineficiente.

```
for (String pal_esp : diccionario.keySet()) {
    System.out.println(diccionario.get(pal_esp);
}
```

NO HACERLO NUNCA ASÍ BAJO PENA DE CATAPULTA.

- Recorrer los dos al mismo tiempo (EntrySet) a los pares claves-valor se les llama entradas:

```
for (Entry<String,String> traduccion: diccionario.entrySet()) {
    System.out.println(traduccion.getKey()+" es "+traduccion.getValue());
}
```

ESTA SI ES LA MANERA ACERTADA

En alguna ocasión podemos necesitar manipular el Hashmap como una colección de clave-valor y aplicar las operaciones que podemos hacerle a un ArrayList como filtrarlos, ordenarlos y buscar el mayor/menor dato según algún criterio que necesitemos. Para ello la forma más compacta es la siguiente:

- Claves como un ArrayList

```
ArrayList<String> claves=new ArrayList<>(diccionario.keySet());
```

- Valores como un ArrayList

```
ArrayList<String> valores=new ArrayList<>(diccionario.values());
```

- EntrySet como un ArrayList

```
ArrayList<Entry<String,String>> entradas=new ArrayList<>(diccionario.entrySet());
```

Un ejemplo muy útil de usar un HashMap puede ser para no necesitar usar estructuras condicionales tipo switch donde se compara un solo valor para mostrarlo como otro, como por ejemplo nos pasa en la cervecería con el tipo

```
HashMap<Character,String> chartipo2String=new HashMap<>();

chartipo2String.put('r',"Rubia");
chartipo2String.put('R',"Roja");
chartipo2String.put('n',"Negra");
chartipo2String.put('t',"Tostada");
```

Aplicado por ejemplo al **toString** de la clase Cerveza

```
        res+="FABRICACION INDUSTRIAL\n";
    }
    res+="TIPO:";
    res+=chartipo2String.get(this.tipo);
    res+="-----------------\n";

    return res;
```

Como es posible que necesitemos traducir un tipo char a su String fuera de la clase podemos ofrecer su funcionalidad (siempre se hace igual) como una constante (no obligatoriamente, pero es lo más recomendable) para ello los pondremos como final static y para poder inicializar una static objeto (de hecho, lo podríamos hacer si llegara el caso). Esto es útil para tener un traductor fijo como por ejemplo los nombres completos de algo sin tener copia en cada objeto ya que no va a variar

```
public final static HashMap<Character,String> chartipo2String=new HashMap<>(){{
    put('r',"Rubia");
    put('r',"Rubia");
    put('R',"Roja");
    put('n',"Negra");
    put('t',"Tostada");

}};
```

Y para usarlo desde fuera de la clase **Cerveza**

```
Cerveza.chartipos2String.get(tipo)
```

No solamente podemos usar como valor tipos básicos, sino que podemos usar cualquier clase de manera que asociamos a un clave (algo único de cada objeto) para asociársela.
Por ejemplo, podemos definir la clase Coche y algunos objetos de dicha clase:

```
public class Coche {
  private String matricula;
  private String marca;
  private double precio;
  ....
```

Creamos un HashMap donde la clave es String y contiene objetos de la clase Coche. El String va a ser la matricula que es única para cada coche.

```
public static void main(){
  Coche coche1= new Coche("9985ABC","Renault",16000);
  Coche coche2= new Coche("6965TRW","Ferrari",200000);
  Coche coche3= new Coche("1233GHX","Toyota",22000);
...

  HashMap<String,Coche> coleccion=new HashMap<>();
```

Para añadir objetos al HashMap usamos el método put estableciendo la clave y el objeto al que va a asociarse:

```
coleccion.put("6965TRW",coche2);
coleccion.put("9985ABC",coche1);
coleccion.put("1233GHX",coche3);
```

Si después queremos encontrar alguno en concreto no tenemos que buscar uno a uno como en una ArrayList, sino que usamos el método get con la clave con la que hayamos guardado con el método put:

```
Coche elferrari=coleccion.get("6965TRW");
System.out.println("Datos del coche:"+elferrari.toString());
```

Esto es muy interesante ya que es una gran mejora a la búsqueda lineal y no hace falta ordenar todo el tiempo para hacer una búsqueda binaria.
Como hemos visto dentro de un HashMap dispondríamos de las mismas funcionalidades que un ArrayList separando la lista de claves y la lista de valores asociados a las claves.