Colecciones

Cuando necesitamos almacenar datos de información pero no sabemos a priori el tamaño que va a ocupar, no podemos usar Arrays en Java (su tamaño se indica al definirlos). Debemos usar unas estructuras llamadas COLECCIONES.

Las colecciones son estructuras dinámicas,es decir, el tamaño crece o se reduce automáticamente según nuestras necesidades (y de forma transparente al programador)

A diferencia de los Arrays simples, JAVA muestra todo la estructura si usamos un Sout para mostrarla: usa un formato predefinido por la consola de Java, normalmente del tipo `[elementos separados por coma ]`

Tipos:

- Listas: sucesión de elementos a los que podemos acceder por su posición. Permiten elementos repetidos.
- Conjuntos o Set: el orden de los elementos no importan, importa si están o no (accedo por el elemento). No permiten elementos repetidos.
- Mapas, diccionarios o arrays asociativos: colección de elementos compuestos por dos elementos: clave y valor.
  - Las claves son identificadores ÚNICOS (no se pueden repetir).
  - Los valores sí pueden repetirse.
  - No importa el orden (accedo por clave)

Nota: En la definición de colecciones vamos a encontrar la notación < T >. Esta quiere decir que dentro de los corchetes angulares va el nombre de una : < Clase > Ej: <String>, <Integer>, <Arma>

ArrayList (listas)

Colección (estructura) que proporciona la implementación de un **array dinámico** => van a crecer o reducirse conforme metemos o destruimos elementos (En los arrays simples no se podían destruir elementos).

Las principales diferencias entre un array normal y un ArrayList son:

- Tamaño: los arrays tienen un tamaño fijo que se especifica al crearlos, mientras que los ArrayList tienen un tamaño dinámico que puede cambiar en tiempo de ejecución.

- Tipo de datos: los arrays solo pueden contener elementos del mismo tipo de datos, mientras que los ArrayList pueden contener elementos de diferentes tipos. Se hace uso del polimorfismo (concretamente se usa la clase Object). Nosotros no vamos a usar esta funcionalidad en el curso.

- Capacidad: en un array normal, su capacidad no puede cambiarse una vez que se crea. En un ArrayList, la capacidad puede aumentar automáticamente según sea necesario para acomodar nuevos elementos.

- Métodos: los arrays tienen una cantidad limitada de métodos que se pueden utilizar para manipular sus elementos, mientras que los ArrayList proporcionan una gran cantidad de métodos para agregar, eliminar, ordenar, buscar y modificar elementos.

Definición:

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

```
//NO PERMITE tipos básicos
ArrayList <T> nombre = new ArrayList <>();
```

Nota: *para usar los tipos básicos de datos es necesario usar sus clases envoltorio: Integer, Double, Boolean, Character...String*

Ejemplos:

```
ArrayList <Integer> lista = new ArrayList <>()
ArrayList <String> lista = new ArrayList <>()
```

Métodos más comunes

`.add(elemento)`	Mete un elemento al final de la lista.
`.add(pos,elem)`	Mete un elemento en la posición indicada desplazando la lista
`.set(pos,elem)`	Sobreescribe el elemento en la posición indicada.
`.size()`	Nos dice el tamaño de la lista.
`.get(pos)`	Devuelve una copia del elemento en la posición indicada no lo elimina)
`.contains(elem)`	Devuelve T o F si el elemento está en la lista o no.
`.indexOf(elem)`	Devuelve la primera posición el elemento indicado.
`.lastIndexOf(ele)`	Devuelve la última posición el elemento indicado.
`.remove(pos)`	Destruye el elemento de la posición indicada y devuelve el elemento
`.remove(objeto)`	Destruye la primera aparción de ese objeto en la lista*
`.clear()`	Se carga todo el contenido de la lista (no la lista)
`.isEmpty()`	Nos dice si la lista tiene algún elemento o no (T o F)

Hay muchos más (tanto comunes para otras colecciones como específicos de ArrayList) pero para nosotros estos son suficientes.

** ej: `lista.remove((Integer)7)` Eso no es lo mismo que `lista.remove(7)`!!!!*

Cómo mostrar el contenido de un arrayList:

```
for(int i=0; i<lista.size(); i++){
      System.out.print(lista.get(i)+" ");
 }
```

Esta forma es INEFICIENTE por culpa del `.size()` Ese método ya realiza bucles para contar el número de elementos de la estructura y nosotros lo invocamos en cada vuelta.

Mejor:

```
int tam = lista.size();
for(int i=0; i<tam; i++){
      System.out.print(lista.get(i)+" ");
 }
```

Aún así, la mejor forma de recorrer colecciones es usando **FOREACH**

FOR-EACH

Bucle FOR mejorado y utilizado para recorrer colecciones:

```
for (tipo variable : coleccion){
      //codigo
}
```

- La cabecera necesita una colección que exista y una variable del mismo tipo que los elementos de la colección.
- Es automático: empieza por el primer elemento y acaba en el último.
- En cada vuelta: la variable se llenará con un elemento de la colección y se ejcuutará el código indicado.

Ejemplo en un arrayList:

```
List<Integer> lista = new ArrayList<Integer>();
lista.add(1);
lista.add(2);
lista.add(3);

for (int num : lista) {
    System.out.println(num); //si fuera objeto se usa toString
}
```

Ejemplo para SET (igual que el anterior cambiando la colección).

En **Mapas** no se puede usar un FOR-EACH de forma sencilla[1] (hay que hacer uso de varios métodos de la estructura). Por este motivo se recomienda usar ITERADORES en este tipo de colección.

1 Hay un ejemplo en el apartado correspondiente a MAPAS más adelante.

SET (conjuntos)

Colección (estructura) no ordenada que solo posee elementos únicos. Se usa cuando necesitamos almacenar valores únicos y no queremos preocuparnos de implementar esa validación al insertar.

Podemos definir esta estructura como “un saco” dónde vamos metiendo elementos que no pueden repetirse. El saco va a crecer o reducirse conforme metemos o destruimos elementos. Si intento meter un valor repetido, no se mete en el saco. No da error.

Existe varias implementaciones que ordenan los elementos automáticamente, tienen más eficiencia en gestión de memoria… nosotros usaremos HashSet

Definición:

```
import java.util.HashSet;
```

```
//NO PERMITE tipos básicos
Set <T> nombre = new HashSet <>();
```

Nota: *para usar los tipos básicos de datos es necesario usar sus clases envoltorio: Integer, Double, Boolean, Character...String*

Ejemplos:

```
HashSet<String> set = new HashSet<>();
HashSet<Integer> set = new HashSet<>();
HashSet<Persona> set = new HashSet<>();
```

Métodos más comunes

`.add(elemento)`	añade el elemento en la estructura si no existe ya. Devuelve booleano con el resultado de la acción.
`.size()`	devuelve entero indicando el número de elementos de la estructura.
`.contains(elemento)`	devuelve un booleano indicando si el elemento está en la estructura.
`.remove(elemento)`	intenta eliminar el elemento de la estructura. Devuelve un booleano el resultado de la acción.
`.removeAll(coleccion)`	Elimina del conjunto todos los elementos que coinciden con los elementos de la colección
`.retainsAll(coleccion)`	Elimina del conjunto todos los elementos que NO coinciden con los elementos de la colección (mantiene los iguales a la colección)
`.clear()`	Se carga todo el contenido de la estructura.

`.isEmpty()`	devuelve booleano indicando si la estructura está vacía o no.

Debido a la naturaleza de esta colección (valores únicos en los que no importa el orden), **no hay método `.get()`**. Si quiero "sacar" un elemento, uso `.contains` para ver si está ese elemento (por lo que ya lo tengo fuera, no?) y si quiero mostrar todos los elementos, uso un foreach.

Para recorrer un SET solo puedo usar FOREACH *(o iteradores)*

```
for(Integer elemento: bolsa){
   System.out.println(elemento);
}
```

Mapas o Diccionarios

Los elementos de esta estructura se llaman ENTRADAS y se componen siempre de dos valores:

1. Clave: valor único en la estructura. Es lo que se usa para acceder a un elemento.
2. Valor: el elemento en sí. Se puede repetir dentro de la estructura por que los valores van asociados a las claves y, por tanto, serían elementos diferentes.

Podemos definir esta estructura como una "agenda" dónde para cada persona (única) guardamos sus datos. La agenda va a crecer o reducirse conforme metemos o eliminamos personas. Si intento meter una clave repetida DA ERROR.

No importa el orden porque para consultar un elemento preguntamos por su clave (que es única).

En JAVA tenemos HashMap, TreeMap (ordena los elementos) y LinkedHashMap. Nosotros usaremos el primero.

Definición:

```
import java.util.HashMap;
```

```
//NO PERMITE tipos básicos
HashMap <T,K> nombre = new HashMap <>();
```

Nota: *para usar los tipos básicos de datos es necesario usar sus clases envoltorio: Integer, Double, Boolean, Character...String*

Ejemplos:

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

Métodos más comunes

`.put(clave, valor)`	Mete una ENTRADA (clave - valor) en el mapa.
`.size()`	devuelve entero indicando el número de elementos de la estructura.
`.get(clave)`	Devuelve el elemento de la clave indicada (no lo elimina)
`.containsKey(clave)`	Devuelve T o F si esa clave está en la estructura o no.

`.containsValue(clave)`	Devuelve T o F si ese valor está en la estructura o no.
`.remove(clave)`	Destruye el elemento de la posición indicada.
`.clear()`	Se carga todo el contenido de la estructura.
`.isEmpty()`	Nos dice si la estructura tiene algún elemento o no (T o F)

Hay que ser consciente de que cada elemento del mapa se compone de CLAVE y VALOR, por lo que es interesante conocer los siguientes métodos:

- `.keySet()`: Devuelve un conjunto (Set, no HashSet) con todas las claves del Map.
- `.values()`: Devuelve una colección (Collection) con todos los valores del Map.
- `.entrySet()`: Devuelve un conjunto (Set, no HashSet) con todos los pares clave-valor.

```
import java.util.HashMap;
import java.util.Set;
import java.util.Collection;

//...

HashMap<String, Integer> edadPorNombre = new HashMap<>();
edadPorNombre.put("Juan", 25);
edadPorNombre.put("María", 30);
edadPorNombre.put("Pedro", 35);

// Obtener una lista de claves
Set<String> claves = edadPorNombre.keySet();
System.out.println("Claves: " + claves);

// Obtener una lista de valores
Collection<Integer> valores = edadPorNombre.values();
System.out.println("Valores: " + valores);

//Obtener conjunto de pares clave-valor
Set<HashMap.Entry<String,Integer>> pares = edadPorNombre.entrySet();
System.out.println("Pares: " + pares);
```

```
Claves: [María, Pedro, Juan]
Valores: [30, 35, 25]
Pares: [María=30, Pedro=35, Juan=25]
```

Recorrer un MAP usando FOR-EACH:

```
Map<String, Integer> mapa = new HashMap<>();
mapa.put("manzanas", 5);
mapa.put("peras", 3);
mapa.put("bananas", 2);

for (Map.Entry<String, Integer> entrada : mapa.entrySet()) {
    String fruta = entrada.getKey();
    int cantidad = entrada.getValue();
    System.out.println("Tengo " + cantidad + " " + fruta);
}
```

No es algo trivial, ya que, debimos usar los tres métodos vistos anteriormente.

Es mucho mejor usar ITERADORES.

Iteradores

Los iteradores son objetos utilizados en Java para recorrer secuencias de elementos, como las colecciones. **Proporcionan una forma estándar, segura y eficiente de acceder a los elementos de una colección uno por uno, sin necesidad de conocer la estructura interna de la colección** (ni su implementación subyacente)**.**

Cuándo es mejor usar un iterador en lugar de un bucle for o foreach:

- Modificación segura durante el recorrido: Si necesitas modificar la colección mientras la estás recorriendo (por ejemplo, eliminar elementos), es más seguro y conveniente usar un iterador. Los iteradores proporcionan métodos como `remove()` que permiten eliminar elementos de la colección durante el recorrido sin lanzar excepciones de concurrencia.

- Flexibilidad: Los iteradores proporcionan más control sobre el recorrido de la colección. Por ejemplo, puedes detener el recorrido en cualquier punto sin necesidad de recorrer la colección completa. Esto puede ser útil en situaciones donde solo necesitas procesar una parte de la colección.

- Su uso es obligatorio en mapas y conjuntos porque son estructuras que no tienen orden, por lo que no se puede usar un FOR o un FOREACH para recorrerlas.

- Compatibilidad con todos los tipos de colecciones: Los iteradores funcionan con cualquier tipo de colección en Java, incluyendo listas, conjuntos, mapas, etc. Por lo tanto, son útiles cuando necesitas escribir código genérico que funcione con diferentes tipos de colecciones.

Por otro lado, los bucles **for y foreach son más simples** y convenientes de usar en muchos casos. Son ideales cuando solo necesitas recorrer la colección y realizar operaciones simples en cada elemento, sin necesidad de modificar la colección mientras se recorre.

Hay que importar: **`import java.util.Iterator;`**

```
Iterator< tipo elemento> nombre = estructura.iterator()
```

EJEMPLO: `Iterator<Integer> it = lista.iterator();`

`it.hasNext()`	Devuelve T o F si hay elementos por recorrer dentro d ella estructura
`it.next()`	Devuelve el siguiente elemento a recorrer.
`it.remove()`	Elimina el elemento por el que va el iterador.

Borrar elementos mientras recorro una colección:

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

//ESTO FALLA!!!
for(Integer ele: lista){
    if(ele%2 == 0){
       lista.remove(ele);
   }
}
//En un ArrayList podemos usar un FOR pero ¿y si tengo un SET?

//El FOR de esta forma falla (aunque es buena practica el TAM)
int tam = lista.size();
for (int i = 0; i < tam; i++) {
    if (lista.get(i)%2 == 0){
        System.out.println("Eliminando..."+lista.get(i));
        lista.remove(i);
    }
}
```

Solución:

```
Iterator<Integer> it = lista.iterator();

while (it.hasNext()){
    int ele = it.next();
    if (ele % 2 == 0){
       it.remove();
    }
}
```

Supongamos un SET con muchos elementos:

```
HashSet<Integer> bolsa = new HashSet<>();
```

podemos decidir cuantas iteraciones damos usando iteradores:

```
Iterator<Integer> iterador = conjunto.iterator();
int contador = 0;

while (iterador.hasNext() && contador < 3) {
    Integer elemento = iterador.next();
    System.out.println(elemento);
    contador++;
}
```

Recorrer un MAP usando un iterador:

```
Map<String, Integer> calificaciones = new HashMap<>();
calificaciones.put("Juan", 8);
calificaciones.put("María", 7);
calificaciones.put("Pedro", 9);
calificaciones.put("Luisa", 6);
calificaciones.put("Ana", 10);

Iterator<Map.Entry<String, Integer>> iterador =
calificaciones.entrySet().iterator();

while (iterador.hasNext()) {
    Map.Entry<String, Integer> entrada = iterador.next();
    String nombre = entrada.getKey();
    int calificacion = entrada.getValue();
    System.out.println("Nombre: " + nombre + ", Nota: " +
calificacion);
}
```