{ Programación }

1º DAM

COLECCIONES

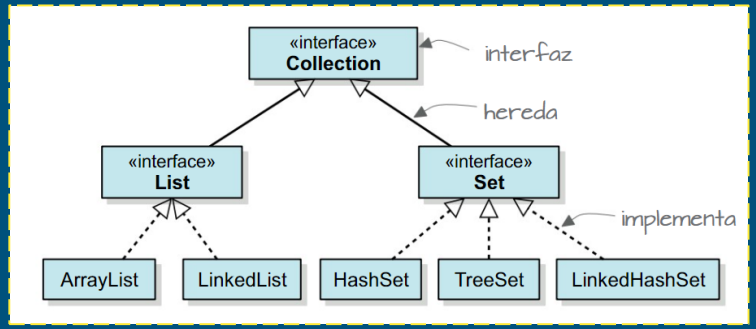
CONTENIDOS

- Interfaz Collection
- Métodos específicos de la interfaz List
- Interfaz Set
- Clase Collections
- Interfaz Map

1. Introducción

Una **colección** es un objeto con un conjunto dinámico de elementos, es decir, el número de elementos puede crecer y decrecer. Son objetos creados para una lógica en la que no se sabe el número de elementos que va a tener que almacenar. Para este fin, Java nos proporciona una serie de estructuras dinámicas que comparten un conjunto de métodos declarados en la interfaz **Collection**. Todas ellas implementan dicha interfaz, aunque de distinta forma

Existen tres tipos generales de colecciones: conjuntos, listas y mapas; los interfaces **Set , List y Map** describen las características generales de éstos.



1. Introducción

Hay colecciones de diferentes tipos, por eso no existe una clase colección, sino todo un marco de trabajo que permiten la manipulación de los datos almacenados en las colecciones. Los tipos fundamentales son tres:

- **Listas**: colección ordenada de elementos que permite duplicados. Implementan la interfaz List que, hereda de Collection.
- **Conjuntos**: presenta una colección de elementos únicos, es decir, no permite duplicados. Implementan la interfaz Set, que también hereda de Collection.
- **Mapas**: están dentro de las colecciones, aunque no implementan la interfaz Collection. Representa una colección de pares clave-valor donde cada clave está asociada a un valor. Las claves son únicas, pero los valores pueden duplicarse . Los mapas implementan la interfaz Map, que no hereda de Collection.

2. Interfaz Collection

La interfaz **Collection** define las funcionalidades comunes a todas las colecciones, ya sean **listas** o **conjuntos**. Sin embargo, las clases que la implementan son listas o conjuntos, es decir, implementan la interfaz List o Set, que son extensiones de Collection. Ninguna implementa Collection directamente. Por ello, para poner ejemplos prácticos de ella, tendremos que usar listas o conjuntos.

Llamaremos colección a toda instancia de alguna de las clases que implementan la interfaz Collection. Estas incluyen: **ArrayList, LinkedList, HashSet, TreeSet y LinkedHashSet**. Los mapas (HashMap, TreeMap y LinkedHashMap) no son colecciones aunque guardan relación con ellas.

3. Listas

Las listas son clases que implementan la interfaz List, y sirven para almacenar datos que se pueden repetir y cuyo orden de inserción puede ser relevante. Hay dos implementaciones de List, las clases ArrayList y LinkedList. Las dos proporcionan los mismos métodos y funcionalidades. La diferencia entre ArrayList y LinkedList radica en la implementación interna y solo afecta levemente al rendimiento.

La declaración de un ArrayList se hace según el siguiente formato.

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

3. Listas

O bien, de forma más general, dado que tanto la clase ArrayList como LinkedList implementan todos los métodos de la interfaz List, es posible utilizar una variable de este tipo para referenciar objetos de ambas clases.

```
List<E> lista = new ArrayList<>{};
```

En esta lista solo se podrán insertar objetos (elementos) del tipo E.

La construcción de un ArrayList que nos sirva para guardar objetos de Cliente, será:

```
List<Cliente> listaClientes = new ArrayList<>{};
```

Una vez creada la colección listaClientes, disponemos de una estructura dinámica donde insertar o eliminar objetos Cliente. Antes vamos a definir una clase Cliente que nos permita probar los distintos métodos con ejemplos concretos:

3. Listas

```
class Cliente implements Comparable<Cliente>{
        String dni;
        String nombre;
        LocalDate fechaNacimiento;

        Cliente(String dni, String nombre, LocalDate fechaNacimiento){
                this.dni=dni;
                this.nombre=nombre;
                this.fechaNacimiento=fechaNacimiento;
        }

        public boolean equals(Object ob){
                return dni.equals(((Cliente) ob).dni);
        }
        public int compareTo(Cliente otro){
                return dni.compareTo(otro.dni);
        }
        public String toString(){
                return "DNI: "+dni+ " Nombre: "+nombre;
        }
}
```

3. Listas

Para estudiar la interfaz Collection, dado que la interfaz List hereda de ella, vamos a referenciar la lista listaClientes con la variable coleccionClie del tipo Collection. Con ello conseguiremos tener acceso únicamente a las funcionalidades de Collection, que son las que vamos a estudiar a continuación, y no a las específicas de la interfaz List, que estudiaremos más adelante.

```
Collection<Cliente> coleccionClie = listaClientes;
```

4. Métodos básicos de la interfaz Collection

Método de inserción

Es aquel que sirve para añadir elementos nuevos en una colección.

- boolean add(E elem): se le pasa el objeto que se va a insertar. Si la inserción tiene éxito, devuelve true. En caso contrario, false. En general, es común que un método devuelva true cuando, al ejecutarse, cambia la estructura de una y false si la colección queda inalterada. Si la colección es una lista, el nuevo elemento siempre se insertará, y además lo hará al final. En cambio, como veremos más adelante, los conjuntos será distinto.

Creamos un nuevo objeto de la clase cliente y lo añadimos a la colección collecionClie:

```
Cliente cliente = new Cliente("11134567M", "Alba", "12/02/2000");
coleccionClie.add(cliente);
```

4. Métodos básicos de la interfaz Collection

Método de eliminación

- boolean remove (Object ob): elimina un elemento ob de una colección. Si está repetido, elimina solo el primero que encuentra. Devuelve true si la eliminación ha tenido éxito y false en caso contrario, por ejemplo, si el objeto no estaba en la colección.

  coleccionClie.remove(cliente);

- void clear() : nos permite eliminar todos los elementos de una colección y dejarla vacía. Esto no significa eliminar la propia colección, del mismo modo que vaciar una bolsa de caramelos no significa destruir la bolsa. La colección, simplemente, queda vacía y disponible para volver a insertar nuevos elementos.

  coleccionClie.clear();

4. Métodos básicos de la interfaz Collection

Método de comprobación

Nos permiten comprobar el estado de una colección. Como hemos dejado la colección vacía, vamos a empezar insertando algunos elementos para seguir experimentando:

```
coleccionClie.add(new Cliente("111", "Marta", "12/02/2000"));
coleccionClie.add(new Cliente("115", "Jorge", "16/03/1999"));
coleccionClie.add(new Cliente("112", "Carlos", "01/10/2002"));
```

- **int size( ):** nos permite saber, en cada momento, el número de elementos insertados en una colección. Por ejemplo,
  ```
  coleccionC1ie.size(); //devuelve 3
  ```
- **boolean isEmpty():** permite saber si una colección está vacía. Devuelve true si está vacía y false en caso contrario.
  ```
  coleccionC1ie.isEmpty() ; //devolverá false
  ```
- **boolean contains (Object ob**): nos dice si un elemento ob determinado está en una colección. Devuelve true si ob pertenece a la colección y false en caso contrario. En nuestro ejemplo,
  ```
  coleccionC1ie.contains (new Cliente ("115" , "Jorge" , "16/03/1999")); //true
  ```

5. Métodos globales de la interfaz Collection

Hasta ahora hemos visto métodos de las colecciones que afectan a un solo elemento. Existen otros métodos, llamados *métodos globales*, en los que intervienen más elementos, incluso más de una colección.

- **boolean containsAll(Collection<?> c):** se le pasa como parámetro otra colección. Devuelve true si todos los elementos de c están en la colección que hace la llamada y false si hay al menos un elemento de c que no.
- **boolean addAll(Collection<? extends E> c):** añade a la colección que hace la llamada todos los elementos de la colección c. Si es una lista, se añadirán todos al final, aunque estén repetidos.
- **boolean removeAll(Collection<?> c):** elimina de la colección invocante todos los elementos que estén contenidos en c. Después de ejecutar el método no habrá elementos comunes a las dos colecciones.

6. Métodos específicos de la interfaz List

Todos los métodos vistos hasta ahora pertenecen a la interfaz Collection y, aunque los hemos probado con listas, son implementados por todas las colecciones, tanto listas como conjuntos. En realidad, las listas implementan la interfaz List, que hereda de Collection, añadiéndole una serie de métodos y funcionalidades específicas, que no comparten los conjuntos.
La funcionalidad más importante exclusiva de las listas (ya sean de la clase ArrayList como de LinkedList) es el acceso posicional a sus elementos por medio de índices. El primer elemento tiene índice 0, el segundo índice 1 y así sucesivamente, como en las tablas.

```
List<Integer> listaEnteros = new ArrayList<>();

listaEnteros.add(3);
listaEnteros.add(1);
listaEnteros.add(-2);
listaEnteros.add(0);
listaEnteros.add(3);
listaEnteros.add(7);

System.out.println(listaEnteros);//[3, 1, -2, 0, 3, 7]
```

6. Métodos específicos de la interfaz List

Los métodos más importantes aportados por la interfaz List son:

- E get(int indice): devuelve el elemento que ocupa el lugar indice en la lista.
- E set(int indice, E elem): guarda el elemento elem en la posición indice, machacando el valor que hubiera previamente en esa posición, que es devuelto.
- void add(int indice, E elem): inserta el valor elem en la posición indice. Todos los elementos que ocupaban una posición igual o mayor que indice, se desplazan una posición hacia el final de la lista, para dejar hueco al nuevo elemento.
- boolean addAll(int indice, Collection c): inserta todos los elementos de la colección c, en el mismo orden que tengan, en la lista que invoca al método, empezando por el lugar indice y desplazando hacia el final todos los elementos de la lista original a partir de indice, incluido este, tantos lugares como sean necesarios. Los elementos de la colección deben ser del mismo tipo E que los de la lista original, o de un subtipo de E.
- E remove(int indice): elimina el elemento que ocupa el lugar indice y lo devuelve.

6. Métodos específicos de la interfaz List

Además de los métodos de lectura, escritura, inserción y eliminación de elementos, heredados de la interfaz Collection, la interfaz List añade funciones de búsqueda, ordenación y comparación.

- int indexOf (Object ob) : devuelve el índice de la primera ocurrencia de ob en la lista. Si no está, devuelve -1.
- int lastIndexOf (Object ob): hace lo mismo que indexOf() pero empezando la búsqueda por el final, devolviendo la última ocurrencia de ob.
- boolean equals (Object otraLista): compara dos listas, tanto si las dos son ArrayList como si son LinkedList, o una de cada, y devuelve true si ambas tienen exactamente los mismos elementos, incluidas las repeticiones, en el mismo orden.
- void sort (List Lista, Comparator c) : ordena la lista invocante con el criterio de c, cuya implementación compara objetos de la clase.

7. Interfaz Set

La interfaz Set trata los datos como un conjunto matemático, eliminando las repeticiones y sin un orden preestablecido.
Todos sus métodos los hereda de Collection. Lo único que añade es la restricción de no permitir duplicados. Esto significa que si intentamos insertar un elemento que ya existe, no lo hará.
El conjunto de métodos disponibles es el mismo que vimos en los apartados de métodos básicos y globales de las colecciones:

```
int size()
boolean isEmpty()
boolean contains (Object element)
boolean add(E element)
boolean remove (Object element)
Iterator<E>iterator ( )
boolean containsAll (Collection<?>c)
boolean addAll (Collection<? extends E>c)
boolean removeAll (Collection<?>c)
boolean retainA11 (Collection<?>c)
void clear()
Object [] toArray ()
<T> T[] toArray( T[])
```

7. Interfaz Set

Las diferencias más importantes son el orden en que se van insertando los elementos nuevos y que un elemento que ya está en el conjunto no se puede volver a insertar, ya que no son posibles los elementos repetidos. Cuando intentemos insertar un elemento repetido con el método add() o con addAll () , no se producirá ningún error ni se arrojará ninguna excepción; sencillamente, el elemento no se inserta y el método devuelve false.

Las implementaciones de Set son las clases: HashSet, TreeSet y LinkedHashSet.

- HashSet: tiene un buen rendimiento, aunque no garantiza ningún orden en la inserción.
- TreeSet: a pesar de tener peor rendimiento, Mantiene los elementos en un orden naturalmente ordenado
- LinkedHashSet: inserta los elementos al final, con lo cual se garantiza un orden basado en la inserción.

Rendimiento: HashSet > LinkedHashSet > TreeSet

7. Interfaz Set

Por ejemplo, vamos a declarar un conjunto de clientes:

```
TreeSet<Cliente> conjuntoCliente = new TreeSet<>();

conjuntoCliente.add(new Cliente("111", "Marta", "12/02/2000"));
conjuntoCliente.add(new Cliente("115", "Jorge", "16/03/1999"));
conjuntoCliente.add(new Cliente("112", "Carlos", "01/10/2002"));
```

Donde, si hacemos un System.out.println(conjuntoCliente), veríamos por pantalla:

```
[DNI: 111 Nombre: Marta Edad: 20
,DNI: 112 Nombre: Carlos Edad: 18
,DNI: 115 Nombre: Jorge Edad: 21
]
```

Observamos que el orden en que aparecen no coincide con el orden de inserción, sino que están ordenados por DNI creciente.

7. Interfaz Set

Si ahora intentamos volver a insertar uno de los elementos anteriores, por ejemplo, el de Marta,

```
boolean insertado = conjuntoCliente.add(new Cliente ("111" "Marta" , "12/02/2000"));
System.out.println (insertado) ; //false, ya que no se ha insertado
System.out.println (conjuntoCliente) ;
```

aparece por pantalla

```
false
```

donde vemos que la inserción ha devuelto false (no se ha insertado) y que el conjunto no ha cambiado.

8. Clase Collections

Además de los métodos aportados por las interfaces Collection, List y Set, la clase Collections (no confundirla con la interfaz Collection) reúne una serie de utilidades en forma de métodos estáticos que trabajan con tipos genéricos. En ellos, el primer parámetro de entrada es la colección sobre la que deseamos operar y comprende métodos de búsqueda, ordenación y manipulación de datos, entre otros. Casi todos ellos operan sobre listas, aunque algunos valen para cualquier colección.

Esta clase se encuentra en el paquete java.util y está diseñada para trabajar con colecciones

8. Clase Collections

MÉTODOS DE ORDENACIÓN

Ya vimos que las listas se pueden ordenar por medio del método sort(), al que se le pasa un comparador. Sin embargo, la clase Collections también posee métodos sort() estáticos.

static <*T* extends *Comparable*<*? super T*>> *void* sort(*List*<*T*> *lista*). Ordena una lista que se le pasa como argumento. El criterio de ordenación será el que establece el comparador, que se utilizará para determinar el orden de los elementos en la lista.

```
List<Cliente> lista = new ArrayList<>();

lista.add(new Cliente("111", "Marta", "12/02/2000"));
lista.add(new Cliente("115", "Jorge", "16/03/1999"));
lista.add(new Cliente("112", "Carlos", "01/10/2002"));

Collections.sort(lista); //Ordenará por DNI ya que fue el comparador definido en la clase Cliente
```

8. Clase Collections

MÉTODOS DE BÚSQUEDA

Uno de los métodos más importantes de la clase Collections es:

static *int* binarySearch(List lista, Object cl, Comparator cmp): hace una búsqueda binaria de un objeto, llamado clave de búsqueda, en una lista que debe estar ordenada previamente. Todo ello necesita un criterio de ordenación.

Vamos a hacer una búsqueda en la lista de clientes. En primer lugar, la volvemos a ordenar por DNI, que es el orden natural.

```
Collections.sort(lista); //ordenada por dni (orden natural)
```

Al método binarysearch() se le pasan como parámetros la lista en la que queremos hacer la búsqueda y el objeto clave que queremos buscar. Devuelve el índice de este último si lo encuentra. Por ejemplo, si queremos buscar a Carlos, cuyo DNI es «112»,

```
int indice = Collections.binarySearch(lista, new Cliente("112", null, null));
```

9. Recorrer colecciones

Las **enumeraciones**, definidas mediante la interfaz *Enumeration*, nos permiten consultar los elementos que contiene una colección. La enumeración irá recorriendo secuencialmente los elementos de la colección. Normalmente, el bucle para la lectura de una enumeración será el siguiente:

```
while (enum.hasMoreElements()) {
Object item = enum.nextElement();
// Hacer algo con el item leido
}
```

- hasMoreElements(): Este método comprueba si todavía hay más elementos disponibles en la colección para ser leídos. Devuelve true si hay más elementos disponibles, y false si no hay más.
- nextElement(): Este método obtiene el siguiente elemento de la colección. Devuelve el siguiente elemento en la secuencia de la colección y avanza el cursor interno del enumerador al siguiente elemento.

9. Recorrer colecciones

Otro elemento para acceder a los datos de una colección son los ***iteradores***. La diferencia está en que los iteradores además de leer los datos nos permitirán eliminarlos de la colección. Los iteradores se definen mediante la **interfaz Iterator**,

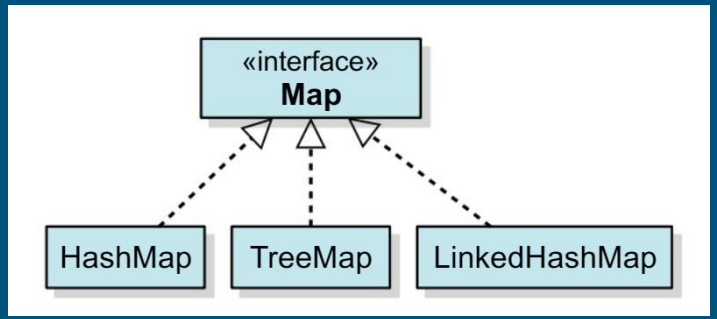
Por ejemplo, podemos recorrer todos los elementos de una colección utilizando un iterador y eliminar aquellos que cumplan ciertas condiciones:

```
while (iter.hasNext()) {
Object item = iter.next();
if(condicion_borrado(item))
iter.remove();}
```

- hasNext(): Este método verifica si hay más elementos en la colección que aún no se han recorrido.
- next(): Este método devuelve el siguiente elemento en la colección.
- Remove(): Este método elimina el último elemento devuelto por next() de la colección

10. Interfaz Map

Los mapas son estructuras dinámicas cuyos elementos son pares clave/valor en vez de valores individuales como en las colecciones. Todas ellas implementan la interfaz Map, que no hereda de Collection. Por tanto, los mapas no son colecciones, aunque están íntimamente relacionados con ellas y funcionan dentro del mismo entorno de trabajo. Vamos a usar tres implementaciones de Map: HashMap, TreeMap y LinkedHashMap, que se diferencian entre sí de forma similar a HashSet, TreeSet y LinkedHashSet.



10. Interfaz Map

- HashMap no garantiza ningún orden específico de los elementos y ofrece un rendimiento constante en tiempo de ejecución para las operaciones básicas (agregar, eliminar, buscar) en el caso promedio. HashMap permite claves y valores nulos.

- LinkedHashMap mantiene el orden de inserción de los pares clave-valor. El rendimiento de LinkedHashMap es ligeramente inferior al de HashMap debido a la sobrecarga adicional para mantener el orden de inserción.

- TreeMap Mantiene los pares clave-valor en un orden naturalmente ordenado (según el método compareTo() de las claves o un Comparator proporcionado). TreeMap no permite claves nulas y ofrece un rendimiento de tiempo logarítmico para las operaciones básicas.

10. Interfaz Map

En un mapa se insertan entradas que constan de una clave, que no se puede repetir, y un valor asociado con ella, que sí puede estar repetido.
Las operaciones fundamentales en un mapa son la inserción, la lectura y la eliminación de entradas.
Para ilustrar el uso de mapas vamos a empezar utilizando la implementación HashMap, que no garantiza ningún orden de inserción de las entradas, aunque es muy eficiente en cuanto a la velocidad de acceso a los datos. El constructor más sencillo es

```
Map<K, V> m = new HashMap<>();
```

donde K es el tipo de las claves y V, el de los valores. Son tipos genéricos que, necesariamente, serán clases o interfaces y no tipos primitivos. Como ejemplo, vamos a suponer que queremos mantener la información de las estaturas de un grupo de escolares, con entradas en las que figura el nombre del alumno (clase String) como clave y la estatura (clase envoltorio Double) como valor,

```
Map<String, Double> m = new HashMap<> ();
```

10 Interfaz Map

Para insertar entradas usamos el siguiente método:
V put (*K clave*, *v valor*): se le pasan como parámetros la clave y el valor asociado con ella. Si no había ninguna entrada previa con la misma clave, se inserta en el mapa la nueva entrada con esa clave y ese valor, y el método devuelve null. Si ya había una entrada con la misma clave, se sustituye el valor antiguo por el nuevo, sin cambiar la clave, y la función devuelve el valor antiguo. Insertemos unas cuantas entradas:

```
m.put("Ana", 1.65);
m.put("Marta", 1.60);
m.put("Luis", 1.73);
m.put("Pedro", 1.69);
```

Con los mapas, igual que con los conjuntos, disponemos también de una implementación de toString ( ) , de forma que podemos visualizarlos *System*.out.println(m); obteniéndose por pantalla,

```
{Marta=1.6, Ana=1.65, Luis=1 73, Pedro=1.69}
```

10. Interfaz Map

Si ahora queremos cambiar la estatura de Pedro, insertamos otra vez un elemento con la misma clave y el nuevo valor

```
m.put("Pedro", 1.71);
```

obteniéndose

{Marta-1.6, Ana=1.65, Luis=1.73, Pedro=1.71}

10. Interfaz Map

- v remove (Object k) : elimina la entrada cuya clave es k, si existe. En este caso, devuelve el valor asociado con esa clave. En caso contrario, devuelve null.
- void clear(): elimina todas las entradas, dejando el mapa vacío.
- V get (Object k) : devuelve el valor asociado con la clave o null si no hay ninguna entrada con esa clave. Por ejemplo, m.get ("Ana") devuelve 1,65.
- boolean containsKey(Object k): devuelve true si hay una entrada con la clave k. Por ejemplo, m.containsKey ("Ana") devolverá true.
- boolean containsValue (Object v) : devuelve true si hay alguna entrada con valor v.
- clear(): Elimina todos los pares clave-valor del mapa.
- V getOrDefault(K key, V defaultValue): se utiliza para recuperar el valor asociado a una clave específica en el mapa. Si la clave no está presente en el mapa, el método devuelve un valor predeterminado especificado.
- Set<Map.Entry<K,V>> entrySet(): Devuelve un objeto Set<Map.Entry<K, V>> que contiene todos los pares clave-valor del mapa como objetos de tipo Map.Entry<K,
- Set<K> keySet(): Devuelve un objeto Set<K> que contiene todas las claves del mapa

10. Interfaz Map

```
Map<String, Double> notasDeAlumnos = new HashMap<>();

// Añadimos las notas de los alumnos
notasDeAlumnos.put("Tim", 9.7);
notasDeAlumnos.put("Bob", 8.5);
notasDeAlumnos.put("Jon", 7.8);
notasDeAlumnos.put("Bob", 8.8);
notasDeAlumnos.put("Bob", notasDeAlumnos.getOrDefault("Bob", 0.0) + 1);   //Bob → 9.8
notasDeAlumnos.put("Jon", notasDeAlumnos.getOrDefault("Kal", 5.0) + 1);   //Jon → 6.0
notasDeAlumnos.put("Kal", notasDeAlumnos.getOrDefault("Bob", 5.0));       //Kal → 9.8
notasDeAlumnos.put("Kal", notasDeAlumnos.getOrDefault("Sam", 0.0));       //Kal → 0.0

// Mostramos datos con entrySet()
System.out.println("Notas alumnos:");
for (Map.Entry<String, Double> pares : notasDeAlumnos.entrySet()) {
    System.out.println("La nota de " + pares.getKey() + " es " + pares.getValue());
}
// Mostramos nombres de los alumnos con keySet() y nota media
System.out.println("Alumnos:"+notasDeAlumnos.keySet()); //Alumnos:[Bob, Kal, Jon, Tim]
double sumaNotas = 0;
for (Double nota : notasDeAlumnos.values()) sumaNotas += nota;
System.out.println("Nota media: " + sumaNotas / notasDeAlumnos.size()); //Nota media: 6.375
```