COLECCIONES en Java

Una **COLECCIÓN** es un grupo de objetos **(elementos)** en memoria. Su uso es muy común para organizar datos en memoria de forma dinámica y gestionarlos en toda su amplitud: añadir|eliminar|consultar elementos, ordenar, buscar, rellenar, mezclar … y en general cualquier otro tipo de manipulación que pueda tener interés realizar sobre un conjunto de elementos de un determinado tipo.

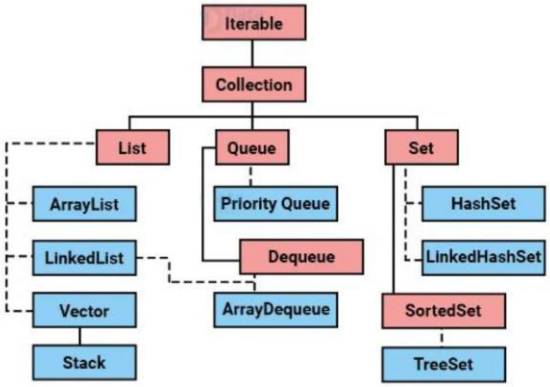
- Nombres de alumnos de una clase
- Marcas de vehículos en un concesionario
- Libros en una biblioteca
- Artículos en una tienda
- Usuarios/Clientes humanos en cualquier aplicación de gestión
- …

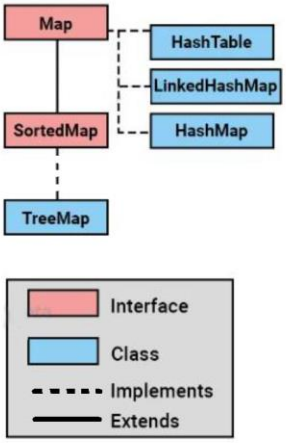
Java dispone de una amplia variedad de **colecciones** con características específicas, para que cada programador/a encuentre siempre la que más se adapta a sus necesidades en un supuesto determinado.

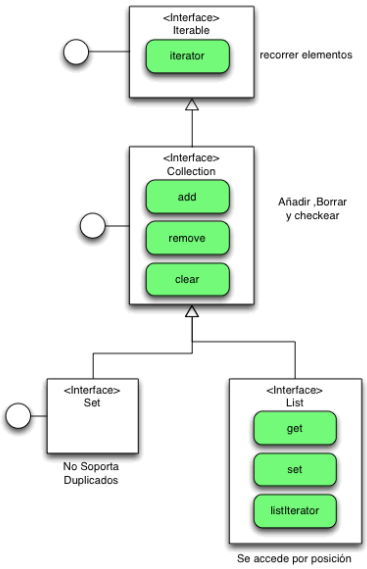
Java proporciona en su **API** la interface **Collection,** como prototipo de la funcionalidad básica que toda colección debe ofrecer. Toda colección que implementa el interface **Collection** de Java está obligada a ofrecer una serie de métodos comunes, entre los que destacan: añadir **add(),** eliminar **remove(),** conocer el tamaño de la colección **size()** … etc.

Collection se ramifica en las subinterfaces **Set, List y Queue**. Para nosotros/as es importante saber que toda colección de tipo List, Set o Queue implementa los métodos de la interface **Collection**. A través de la **herencia,** los métodos declarados en **Collection** se van propagando, y eso facilita el trabajo.

Paralelamente, Java proporciona la interface **Map**, que define un tipo especial de colecciones "indexadas", en los que cada elemento va asociado a una clave-**key** que lo identifica. Los métodos para gestionar **Maps** son sensiblemente diferentes, pero igual de intuitivos y accesibles. En la figura vemos que **Map** no hereda de la interface **Collection**. Los **Maps** son ampliamente utilizados, y trabajaremos con ellos en nuestros ejemplos, al igual que con **Lists** y **Sets**.





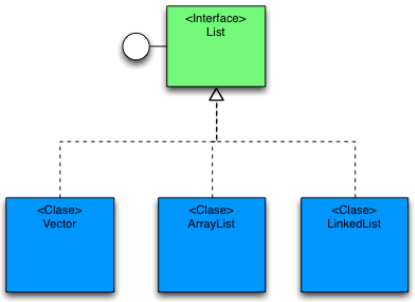


<u>LISTAS:</u> Interface List<E> (E – elemento del Elemento en la lista)

Colecciones muy utilizadas en Java, sobre todo en los primeros pasos en programación, pues nos recuerdan a los **Arrays**. Aportan grandes ventajas sobre los Arrays, pues su creación es dinámica, sin declarar tamaño inicial, y disponen de un conjunto de métodos sencillos para dominar rápidamente su funcionalidad básica.

Métodos principales interface ArrayList:

- boolean **add**(E elemento) *- añade elemento*
- boolean **remove**(E elemento) *- elimina elemento*
- void **clear**() *- elimina todos los elementos*
- int **size**() *– devuelve el número de elementos en la lista*
- void **add**(int posicion, E elemento) *- añade elemento en una posición determinada*
- boolean **contains**(E elemento) *- comprueba si está un elemento*
- boolean **equals**(Object x) *- compara la lista con otra*
- E **get**(int posicion) *- devuelve el elemento de una posición*
- int **indexOf**(E elemento) *- devuelve la posición de un elemento*
- boolean **isEmpty**() *- true si la lista está vacía*
- E **set**(int posicion, E elemento) *- Reemplaza elemento de una posición*
- **remove**(int posición) *- Elimina elemento de una posición.*



Implementaciones (Diferentes tipos de List en Java)

- **Vector<E>**
- **ArrayList<E>**
- **LinkedList<E>**

Métodos específicos LinkedList

- E **getFirst**() *- devuelve el primer elemento*
- E **getLast**() *- devuelve el último elemento*
- E **removeFirst**() *- borra y devuelve el primero*
- E **removeLast**() *- borra y devuelve el último*
- void **addFirst**(E elemento) / **addLast**(E elemento) *- añade un elemento al principio/final*

Ejemplo: el acceso a un elemento determinado de una **List** es como en un Array, a través de un índice de tipo **int** que indica su **posición**. Los elementos de una **List** siempre van desde **(0) – (List.size()-1)**

```
ArrayList<String> lista = new ArrayList() ;
lista.add("hola");
lista.add("adios");
lista.add("bye");
System.out.println(lista.size());          // Devuelve 3
System.out.println(lista.get(2));          // Devuelve bye
for (String s: lista){
    System.out.print(s + "  ");            // Recorrido de la lista con For Each - Imprime: hola adios bye
}

Iterator<String> it = lista.iterator();
while(it.hasNext()) {
    System.out.print(it.next() + "  ");    // Recorrido de la lista con un Iterator - Imprime: hola adios bye
}
```

<u>CONJUNTOS:</u> Interface Set<E>

Colección Java con la característica destacada de **<u>comprobar y evitar las repeticiones de elementos</u>**. Un conjunto es por definición una colección de elementos **NO REPETIDOS.**

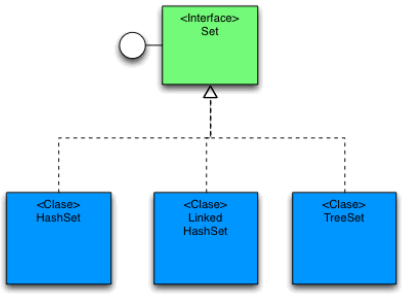
IMPORTANTE: Los Set en Java no disponen de un método **get()** para obtener/acceder a un elemento concreto del **Set**. Si queremos hacer esto podremos fácilmente convertir el Set a un **ArrayList** o **Array**. Entonces, **¿Para qué sirven los Set?** … la respuesta es que los Sets Java se organizan basándose en una tabla **Hash** y esto hace que para colecciones de datos grandes la **eficiencia en búsquedas** de datos sea mucho mayor que con **List**.

Métodos principales interface Set:

- boolean **add**(E elemento) *- añade elemento*
- boolean **remove**(E elemento) - *elimina elemento*
- void **clear**() *- elimina todos los elementos*
- int **size**() *– devuelve el número de elementos en la lista*
- boolean **contains**(E elemento)
- boolean **equals**(Object x)
- boolean **isEmpty**()
- Iterator<E> **iterator()**

Implementaciones (Diferentes tipos de Set en Java)

- **HashSet<E>** (Elementos desordenados)
- **TreeSet<E>** (Elementos ordenados)
- **LinkedHashSet<E>** (Elementos ordenados)



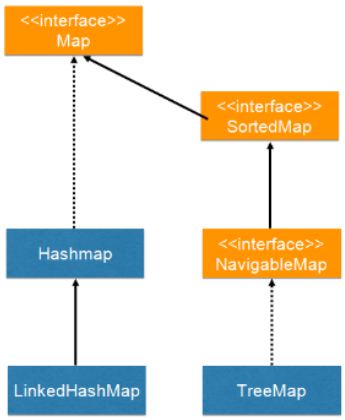
Ejemplo:

```
HashSet<String> conjunto = new HashSet() ;
conjunto.add("hola");
conjunto.add("adios");
conjunto.add("bye");
conjunto.add("bye");                    // Elemento repetido no será admitido en el conjunto
System.out.println(conjunto.size());    // Devuelve 3
for (String s: conjunto){
    System.out.print(s + "  ");         //  Recorrido con For Each (No garantiza el orden de inserción)
}
conjunto.remove("hola");                // Elimina el elemento "hola"

Iterator<String> it = conjunto.iterator();
while(it.hasNext()) {
     System.out.print(it.next() + "  ");   //  Recorrido con Iterator
}

ArrayList<String> lista=new ArrayList(conjunto);   // Convertimos Set en ArrayList. También se puede hacer al revés

for (String s: lista){
    System.out.print(s + "  ");          // recorremos la List creada a partir del Set
}
```

MAPAS (Maps)

Los Mapas son colecciones especiales definidas por una **Interface** propia. Esta interface es implementada por las siguientes clases, entre otras:

- **HashMap**
- LinkedhashMap
- TreeMap
- …



Hay un buen número de clases que implementan la interface **Map** en Java. Cada una tiene sus características, pero es imposible verlas todas en este curso, así que vamos a centrarnos en la **funcionalidad básica definida por la interface Map**, que todas las colecciones de tipo **Map** deben de implementar y ofrecer a los programadores/as. Queda para el alumnado la profundización en la especificidad de cada variante de **Map**.

Interface Map<K,V>

Los mapas Java trabajan con clave-valor **(Key-Value)** pudiendo acceder directamente a elementos **(values)** por su clave **(key)** sin necesidad de especificar la posición. **NO SE ADMITEN VALORES DUPLICADOS EN LA CLAVE.**

Algunos de los métodos comunes de cualquier Map:

- void **clear**()
- boolean **containsKey**(Object clave)
- boolean **containsValue**(Object valor)
- boolean **equals**(Object x)
- V **get**(Object clave)
- boolean **isEmpty**()
- Set<K> **keySet**()
- V **put**(K clave, V value)
- V **remove**(Object clave)
- int **size**()

Implementaciones más usadas:

- HashMap<K,V>
- LinkedHashMap<K,V>
- TreeMap<K,V>
- Hashtable<K,V>

Ejemplo:

```
HashMap<String, String> capitales = new HashMap();
capitales.put("España", "Madrid");
capitales.put("Alemania", "Berlin");
capitales.put("Francia", "Paris");
capitales.put("Italia", "Roma");
capitales.put("Portugal", "Lisboa");

capitales.get("España");                    // devuelve Madrid
```

```
capitales.remove("Italia");                    //elimina de la colección el objeto ("Italia", "Roma")

System.out.println(capitales.size());          // Devuelve 5  -  (4) si hemos ejecutado capitales.remove("Italia")


for (String k: capitales.keySet()) {
    System.out.println(k + "  ");              // Imprime sólo claves - Keys
}

for (String v: capitales.values()) {
    System.out.println(v+ "  ");               // imprime sólo valores - Values
}

for (String k: capitales.keySet()) {
    System.out.println(k + " = " + capitales.get(k));      //   Imprime Claves y valores
}

// con un Iterator
Iterator<String> it = capitales.keySet().iterator();
while( it.hasNext()) {
    String s=it.next();
    System.out.println(s + " = " + capitales.get(s));     //   Imprime Claves y valores
}
```

ITERADORES para recorrer colecciones: Clase Iterator<E>

Se pueden utilizar objetos de tipo **Iterator** asociados a **TODAS** las colecciones vistas anteriormente.

Métodos de un Iterator:

- boolean **hasNext**() *// Devuelve true si aún quedan elementos en la colección*
- E **next**() *// Devuelve el elemento actual y avanza al siguiente elemento de la colección*
- void **remove**() *// Permite eliminar elementos de una colección "al vuelo"*

Ejemplo: uso de **Iterator** en el proyecto Biblioteca para borrar los préstamos de un usuario (a su vez eliminado)

```
Iterator<Prestamo> it = prestamos.iterator();
while(it.hasNext()) {
   if (it.next().getUsuarioPres()==usuarios.get(existeDni)){
      it.remove();
   }
}
usuarios.remove(existeDni);
```

API DE STREAMS – COLECCIONES COMO STREAMS.

Como ya se comentó en el tema 2, cuando hablábamos de Arrays, a partir de java 8 se producen importantes novedades en Java que afectan a múltiples aspectos del lenguaje. **El API Stream** es una de ellas. Las colecciones se pueden convertir a **Streams** y trabajar con ellas como **Streams** empleando sentencias de programación funcional.

Ejemplo: Recorrer una ArrayList de números enteros calculando la suma de los elementos que son >10

```
import java.util.*;
public class EjStream {
  public static void main(String[] args) {
    ArrayList<Integer> nums = new ArrayList(List.of(21,34,6,11,11,11,12,1,3,5,7));

  // MÉTODO TRADICIONAL CON FOR EACH
    int suma=0;
    for ( int n:nums){
            if (n>10){
              suma+=n;
            }
    }
    System.out.println("La suma de elementos mayores de 10 es:  "+ suma);

  // UTILIZANDO UN ITERATOR
     Iterator<Integer> it = nums.iterator();
     suma = 0;
     while (it.hasNext()) {
       int num = it.next();
       if (num > 10) {
         suma += num;
       }
     }
     System.out.println("La suma de elementos mayores de 10 es:  "+ suma);


  // PROGRAMACION FUNCIONAL CON STREAMS
     System.out.println(nums.stream().filter(i -> i > 10).mapToInt(i -> i).sum());
  }
}
```

El resultado en ambos casos es 100

El uso del API Streams es algo al que todo programador/a de Java debe aspirar a medio plazo, pues reduce considerablemente la cantidad de código, genera resultados a través de una serie de operaciones sobre datos sin modificar estos datos, y permite optimizar procesos para reducir el consumo de recursos (como la memoria)

UTILIDADES (métodos static) para Colecciones y Arrays en Java.

Hay un conjunto de operaciones típicas muy habituales en colecciones y Arrays. **Java las ha incluido como métodos estáticos en la clase Collections (**no confundir con la interface Collection**)** y en la clase **Arrays.**

Java.util.Arrays
java.util.Collections - Es una clase de utilidad ("helper") que sirve de apoyo a las clases que implementan la interface **Collection**. Quedan excluidos los **Maps,** pues no implementan la interface **Collection**.

Arrays y **Collections** son 2 clases que nos van a permitir realizar operaciones comunes con Arrays y colecciones de tipo List, Set y Queue, sin necesidad de programarlas nosotros/as. Recordemos que los **métodos estáticos** los invocamos directamente a través del nombre de clase. **Math.pow() - Arrays.sort() - Collections.sort() - Collections.reverse()**

- Ordenar los elementos según un criterio. **sort()**
- Desordenar los elementos. **shuffle()**
- Buscar un elemento en una colección o Array. **binarySearch()**
- Convertir la colección en Array o viceversa. **asList()**
- Voltear la colección o Array. **reverse()**
- Convertir la colección en Stream. **stream()**

Ordenación natural. Método sort.

El método **sort** ordena los elementos según su orden natural. Los tipos "ordenables" de forma natural son los **números** (de menor a mayor), **Strings** (orden alfabético) y **fechas**. El orden será de menor a mayor.

sort existe tanto para **Arrays** como para **Collections** (los **Maps** no están incluidos)

Ordenar un Array

```
Integer[] numeros ={10,3,5,4,8,1};
Arrays.sort(numeros)   //Si lo recorremos para imprimir obtenemos {1,3,4,5,8,10}
```

Ordenar una lista

```
ArrayList<Integer> lista = new ArrayList(List.of(21,34,6,11,11,11,12,1,3,5,7));
Collections.sort(lista); // Si lo recorremos para imprimir obtenemos (1,3,5,7,11,11,11,12,21,34)
```

Insistir una vez más en que los métodos estáticos se invocan a través del nombre de la clase

Ordenación NO natural. Interfaces Comparator/Comparable.

El problema lo encontramos a la hora de ordenar colecciones de objetos, como en los ejemplos que estamos programando este curso: clientes, libros, artículos, usuarios, cuentas …

Si los elementos de una colección son de tipo objeto **no admiten una ordenación "natural"**, dado que son referencias a memoria. En estos casos, hay que **seleccionar/marcar** un atributo de los objetos para que sea el **criterio de ordenación**. Se puede hacer de 2 posibles **formas**:

Forma 1:

Consiste en crear una clase propia que implemente la interface **java.util.Comparator.** La interface **Comparator** nos obliga a reescribir un método llamado **compare** que es dónde vamos a indicar el atributo de los objetos que servirá para compararlos, y en definitiva **ordenarlos**. Como **Ejemplo** vamos a ordenar la colección **artículos** del proyecto **Tienda**. Utilizamos el atributo **idArticulo** (String). Al usar un atributo String el orden natural resultante será **alfabético**.

```
import java.util.Comparator;
class ComparaArticulos implements Comparator <Articulo> {
    @Override
    public int compare (Articulo a1, Articulo a2){
        return a1.getIdArticulo().compareTo(a2.getIdArticulo());
    }
}
```

Para ordenar los elementos de la colección, simplemente se pasa como segundo parámetro del método **sort** una instancia del **Comparator** creado:

```
Collections.sort(articulos, new ComparaArticulos());
```

Forma 2:

Ejemplo: ordenar la colección **clientes** del proyecto **Tienda**. Utilizaremos el atributo **dni** (String). Requiere modificar la clase de los elementos a comparar implementando la interface **Comparable**. Toda clase que implemente la interfaz **Comparable** será "ordenable" y se puede invocar el método **sort** directamente sin indicar un Comparator como en la **forma 1**. Tan sólo estamos obligados a implementar el método **compareTo** en la clase para indicar que atributo establece la ordenación.

```
public class Cliente implements Comparable<Cliente>{

    // código de la clase tal y como lo teníamos

  @Override
  public int compareTo(Cliente c) {
    return this.getDni().compareTo(c.getDni());
  }
 //método compareTo añadido por el IDE, pero adaptado por nosotros para ordenar clientes por el atributo dni
}
```

Ordenar ahora la lista clientes por **dni** es sencillo. Simplemente: **Collections.sort(clientes);**

Las formas 1 y 2 no son excluyentes. Pueden usarse ambas. La forma 2 establece un criterio de ordenación **a nivel de clase**. En el ejemplo anterior, el resultado de **Collections.sort(clientes)** será la colección clientes ordenada por **dni.** Esto no nos impide crear nuestras propias clase comparadoras como en la **forma 1,** para poder ordenar la colección por otros atributos "a demanda". Ahora bien, como atributo de ordenación "de clase" para un **sort()** por defecto hemos establecido el **dni**.

Se deja cómo práctica ordenar según la forma 1 clientes con otros atributos.

¿Se pueden ordenar los Maps?

Como se ha comentado, **Collections** no está disponible para los **Maps**, así que no vamos a poder ordenar mapas con **Collections.sort()**.

Hay **Maps** que si son ordenables, como **TreeMap**, que implementa el **Interface SortedMap (ver gráfico inicial)**.

Los **HashMaps** no son ordenables. Sólo son ordenables los **Map** que implementan la interface **SortedMap.** Si nos vemos en la necesidad de ordenar un **HashMap**, podemos "volcarlo" en un **TreeMap,** que se ordena automáticamente de forma natural por el valor clave ("key") … sin necesidad de que nosotros hagamos nada.

```
//volcado de un HashMap en un TreeMap
TreeMap<String, Articulo> articulosOrdenado = new TreeMap(articulos);     //articulos es un HashMap
for (Articulo a: articulosOrdenado.values()){
    System.out.println(a);
}
```

Dado que la clave -"key" del HashMap Artículos es el **idArticulo**, el TreeMap se ordena automáticamente por idArticulo, y la colección ordenada de artículos que obtendremos por pantalla saldrá ordenada por **idArticulo**.

También se puede convertir a **Stream** el HashMap y aplicarle sentencias de programación funcional. En Java todo se puede hacer, de una u otra forma. **(Se deja esto como práctica para el alumnado)**

Otros métodos estáticos de las clases Collections y Arrays. (NO disponibles para Maps)

Operación	Descripción	Ejemplos
Desordenar una lista.	Desordena una lista, este método no está disponible para arrays.	Collections.shuffle (lista);
Rellenar una lista o array.	Rellena una lista o array copiando el mismo valor en todos los elementos del array o lista. Útil para reiniciar una lista o array.	Collections.fill (lista,elemento); Arrays.fill (array,elemento);
Búsqueda binaria.	Permite realizar búsquedas rápidas en un una lista o array ordenados. Es necesario que la lista o array estén ordenados, si no lo están, la búsqueda no tendrá éxito.	Collections.binarySearch(lista,elemento); Arrays.binarySearch(array, elemento);
Convertir un array a lista.	Permite rápidamente convertir un array a una lista de elementos, extremadamente útil. No se especifica el tipo de lista retornado (no es ArrayList ni LinkedList), solo se especifica que retorna una lista que implementa la interfaz java.util.List.	List lista=Arrays.asList(array); Si el tipo de dato almacenado en el array es conocido (Integer por ejemplo), es conveniente especificar el tipo de objeto de la lista: List<Integer>lista = Arrays.asList(array);
Convertir una lista a array.	Permite convertir una lista a array. Esto se puede realizar en todas las colecciones, y no es un método de la clase Collections, sino propio de la interfaz Collection. Es conveniente que sepas de su existencia.	Para este ejemplo, supondremos que los elementos de la lista son números, dado que hay que crear un array del tipo almacenado en la lista, y del tamaño de la lista: Integer[] array=new Integer[lista.size()]; lista.toArray(array)
Dar la vuelta.	Da la vuelta a una lista, poniéndola en orden inverso al que tiene.	Collections.reverse(lista);