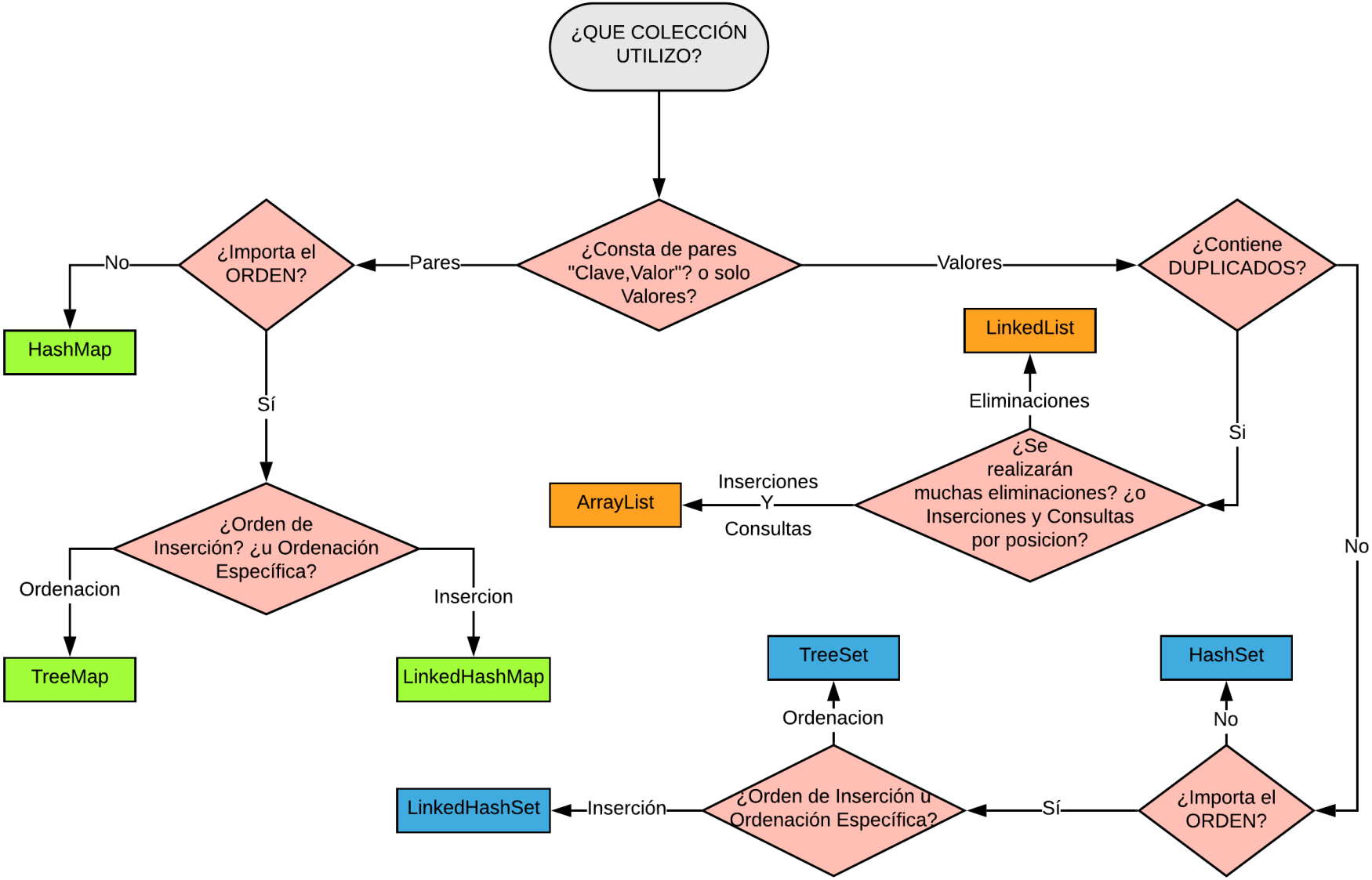
COLECCIONES-JAVA

Podemos almacenar cualquier tipo de objeto y podemos usar una serie de métodos comunes, como pueden ser: añadir, eliminar, obtener el tamaño de la colección...

<table>
<tr><th colspan="2">LIST (Listas)</th><th colspan="2">SET (Conjuntos)</th><th colspan="2">MAP (Mapas)</th></tr>
<tr><td colspan="2">-La interfaz List sí admite elementos duplicados.
-A parte de los métodos heredados de Collection, añade:

-Acceso posicional a elementos: manipula elementos en función de su posición en la lista.
-Búsqueda de elementos: busca un elemento concreto de la lista y devuelve su posición.
-Iteración sobre elementos: mejora el Iterator por defecto.
-Rango de operación: permite realizar ciertas operaciones sobre rangos de elementos dentro de la propia lista.</td><td colspan="2">-Set no puede contener elementos duplicados.
-Contiene, los métodos heredados de Collection.
-Para comprobar si los elementos son elementos duplicados o no lo son, es necesario que dichos elementos sean comparables o tendremos que crear un comparador</td><td colspan="2">-La interfaz Map asocia claves a valores. Esta interfaz no puede contener claves duplicadas y; cada una de dichas claves, sólo puede tener asociado un valor como máximo.</td></tr>
<tr><td colspan="2"></td><td>TreeSet</td><td>Almacena los elementos ordenándolos en función de sus valores.. Los elementos almacenados deben implementar la interfaz Comparable.
Puede personalizarse el comparador, o crear uno si el objeto no es comparable . Luego solo tenemos que pasarselo como parámetro al inicializar el TreeSet.</td><td>TreeMap</td><td>Almacena las claves ordenándolas en función de sus valores..Las claves almacenadas deben implementar la interfaz Comparable.</td></tr>
<tr><td colspan="2"></td><td colspan="2">TreeSet<OBJ> nom = new TreeSet<>(CompPersonlzado);
TreeSet<OBJ> nom = new TreeSet<>(); //<--Sin comparador</td><td colspan="2">TreeMap<String,Integer> nom = new TreeMap< String ,Integer>();</td></tr>
<tr><td>ArrayList</td><td>Se basa en un array redimensionable que aumenta su tamaño según crece la colección de elementos</td><td>HashSet</td><td>Almacena los elementos en una tabla hash. Es la implementación con mejor rendimiento de todas pero no garantiza ningún orden a la hora de realizar iteraciones.</td><td>HashMap</td><td>Esta implementación almacena las claves en una tabla hash. Es la implementación con mejor rendimiento de todas pero no garantiza ningún orden a la hora de realizar iteraciones.</td></tr>
<tr><td colspan="2">ArrayList<OBJ> nom = new ArrayList<OBJ>();</td><td colspan="2">HashSet<OBJ> nom = new HashSet<>();</td><td colspan="2">HashMap<String,Integer> nom = new HashMap< String ,Integer>();</td></tr>
<tr><td>LinkedList</td><td>Esta implementación se basa en una lista doblemente enlazada de los elementos, teniendo cada uno de los elementos un puntero al anterior y al siguiente elemento.</td><td>LinkedHashSet</td><td>Almacena los elementos en función del orden de inserción</td><td>LinkedHashMap</td><td>Esta implementación almacena las claves en función del orden de inserción. Es, simplemente, un poco más costosa que HashMap.</td></tr>
<tr><td colspan="2">LinkedList<OBJ> nom = new LinkedList<OBJ>();</td><td colspan="2">LinkedHashSet<OBJ> nom = new LinkedHashSet<OBJ>();</td><td colspan="2">LinkedHashMap<Byte,Char> L= new LinkedHashMap<Byte,Char>();</td></tr>
</table>

¿QUÉ COLECCIÓN DEBERÍA UTILIZAR?



METODOS DE LA CLASE COLLECTION

COMUNES A TODAS LAS COLECCIONES

INTERFACES QUE IMPLEMENTA	java.util.Collection

TIPO	METODO	DESCRIPCIÓN
int	**size()**	Retorna el número de elementos de la colección.
boolean	**isEmpty()**	Retornará verdadero si la colección está vacía.
boolean	**contains (Object element)**	Retornará verdadero si la colección tiene el elemento pasado como parámetro
boolean	**add(E element)**	Permitirá añadir elementos a la colección.
boolean	**remove (Object element)**	Permitirá eliminar elementos de la colección.
Iterator<E>	**iterator()**	Permitirá crear un iterador para recorrer los elementos de la colección.
Object[ ]	**toArray()**	Permite pasar la colección a un array de objetos tipo Object.
	containsAll(Collection<?> c)	Comprueba si una colección contiene los elementos existentes en otra colección, si es así, retorna verdadero.
	addAll (Collection<? extends E> c)	Permite añadir todos los elementos de una colección a otra colección, siempre que sean del mismo tipo (o deriven del mismo tipo base).
boolean	**removeAll(Collection<?> c)**	Si los elementos de la colección pasada como parámetro están en nuestra colección, se eliminan, el resto se quedan.
boolean	**retainAll(Collection<?> c)**	Si los elementos de la colección pasada como parámetro están en nuestra colección, se dejan, el resto se eliminan.
void	**clear()**	Vacíar la colección.

TIPOS DE COMBINACIONES ENTRE COLECCIONES

(Combinar una colección con otra) **COMUNES A TODAS LAS COLECCIONES**

Combinación.	Código.	Elementos finales del conjunto A.
Unión. Añadir todos los elementos del conjunto B en el conjunto A.	**A.addAll(B)**	Todos los del conjunto A, añadiendo los del B, pero sin repetir los que ya están.
Diferencia. Eliminar los elementos del conjunto B que puedan estar en el conjunto A.	**A.removeAll(B)**	Todos los elementos del conjunto A, que no estén en el conjunto.
Intersección. Retiene los elementos comunes a ambos conjuntos.	**A.retainAll(B)**	Todos los elementos del conjunto A, que también están en el conjunto.

Método SORT()

INTERFACES QUE IMPLEMENTA	java.util.Comparator

+Para ordenar las Clases "NO Comparables", creamos una clase que implemente el Método **Compare** y pasarsela a **sort()**

Creando CLASE Comparador *(Acorde a nuestra necesidad de ordenación)*	**class** *comparadorObjeto* **implements Comparator<***Objeto***>{** **@Override** **public int compare(** Objeto o1, Objeto o2) **{** **return** o1.getNombreObjeto.**compareTo(**o2.getNombreObjeto); **}}**
Utilizamos el Método **sort()** para ordenar la lista.	**Collections.sort(***Objeto***, new** *comparadorObjeto()***);**
++ O si la clase es **"Comparable",** invocar a **SORT** y pasarle **la Lista ++**	
Ordenando: **Sort+"lista"**	**Collections.sort***(lista);*

-SET-

NO AÑADE NINGÚN MÉTODO ADICIONAL *(Solo los que hereda de Collection)*

INTERFACES QUE IMPLEMENTA	**java.util.Set** *(Extiende de "Collection");* ***java.util.TreeSet / LinkedHashSet/ HashSet***

COMPARADORES

Si queremos que "**TreeSet**" ordene la lista y el objeto a comparar NO es Comparable. Tenemos 2 Opciones:

1-CREAR UN **Comparador Personalizado** (CLASE COMPARADOR) **Luego hay que pasar este comparador al TREESET**

```
class comparadorObjeto implements Comparator<Objeto>{
    @Override
    public int compare( Objeto o1, Objeto o2) {
          return o1.getNombreObjeto.compareTo(o2.getNombreObjeto);
    }
}
```

2-IMPLEMENTAR la interfaz **"Comparable<OBJ>" a la Clase que queremos ordenar** (Se ordenaría automaticamente)

```
class Objeto implements Comparable<Objeto>{
    public String codArticulo;
    public String descripcion;
    public int cantidad;

    @Override
    public int compareTo(Objeto o) {
return codArticulo.compareTo(o.codArticulo); }}
```

++ ITERADORES (Set) ++

Necesitamos ITERADORES para acceder a los elementos almacenados en los conjuntos. (podemos usar for-each o Iterator)

+ITERADOR FOR-EACH+

→ (Utilizar SOLO Para LEER)(NO BORRAR ELEMENTOS CON ESTE BUCLE)
Es el más sencillo de implementar (recomendable)

Bucle **FOR-EACH**

```
for (Integer i: conjunto) {
    System.out.println("Elemento almacenado:"+i);  }
```

+"ITERATOR" (Tradicional)+

→ (UTILIZAR ESTE ITERADOR Y SU MÉTODO "REMOVE" EN CASO DE QUERER REMOVER ELEMENTOS)
Nos proporciona 3 MÉTODOS que debemos utilizar para poder iterar la Lista

TIPO	METODO	DESCRIPCIÓN
boolean	**hasNext()**	Retornará true si le quedan más elementos a la colección por visitar. False en caso contrario.
E	**next()**	Retornará el siguiente elemento de la colección, si no existe siguiente elemento, lanzará una excepción (NoSuchElementException para ser exactos), con lo que conviene chequear primero si el siguiente elemento existe.
	remove()	Elimina de la colección el último elemento retornado en la última invocación de next (no es necesario pasarselo por parámetro). **Cuidado, si next no ha sido invocado** todavía, saltará una incomoda excepción.

CREANDO ITERATOR (Tradicional)

***ITERADOR + Bucle WHILE + Métodos del Iterador**

```
Iterator<OBJ> it=ListaTreeSet.iterator();  //Creamos el iterador de el Objeto a Iterar.
while (it.hasNext()) {          // Mientras que haya un siguiente elemento, seguiremos en el bucle.
        OBJ ListaTreeSet=it.next();  // Escogemos el siguiente elemento.
        if (ListaTreeSet%2==0) it.remove(); } //Podemos eliminar el elemento extraído de la lista.
```

MAP

Métodos principales de los mapas. *(Añade métodos propios + los de Collection)*

INTERFACES QUE IMPLEMENTA	**java.util.Map; (***Extiende de "Collection");* **java.util.TreeMap / LinkedHashMap/ HashMap**

TIPO	MÉTODO	DESCRIPCIÓN
V	**put (K key, V value);**	Inserta un par de objetos llave (key) y valor (value) en el mapa. Si la llave ya existe en el mapa, entonces retornará el valor asociado que tenía antes, si la llave no existía, entonces retornará null.
V	**get(Object key);**	Obtiene el valor asociado a una llave ya almacenada en el mapa. Si no existe la llave, retornará null.
V	**remove(Object key);**	Elimina la llave y el valor asociado. Retorna el valor asociado a la llave, por si lo queremos utilizar para algo, o null, si la llave no existe.
boolean	**containsKey(Object key);**	Retornará true si el mapa tiene almacenada la llave pasada por parámetro, false en cualquier otro caso.
boolean	**containsValue(Object value);**	Retornará true si el mapa tiene almacenado el valor pasado por parámetro, false en cualquier otro caso.
int	**size();**	Retornará el número de pares llave y valor almacenado en el mapa.
boolean	**isEmpty();**	Retornará true si el mapa está vacío, false en cualquier otro caso.
void	**clear();**	Vacía el mapa.

En los ejemplos, **V** es el tipo base usado para el **valor** y **K** el tipo base usado para la **llave:**

+ITERADOR para MAP+

Para poder utilizar los iteradores, utilizamos el método **keySet** para generar un conjunto con las llaves existentes en el mapa.
El conjunto generado por **keySet** no tendrá el método **add** para añadir elementos al mismo, hay que hacerlo **a través** del **mapa**.

FOR-EACH (dopado)	**for (Integer llave: OBJ.keySet()) {** *// Recorremos el conjunto generado por keySet, contendrá las llaves.* **Integer valor=OBJ.get(llave); }** //Para cada llave, accedemos a su valor si es necesario.

LIST

Métodos principales de las Listas (**ArrayList** y **LinkedList**). (Añaden metodos propios + comunes de la **Clase Collection**)

INTERFACES QUE IMPLEMENTA	**java.util.List**, y dos implementaciones java.util.**LinkedList** / **ArrayList**

TIPO	MÉTODO	DESCRIPCIÓN
E	**get(int index)**	El método get permite obtener un elemento partiendo de su posición (index).
E	**set(int index, E element)**	El método set permite cambiar el elemento almacenado en una posición de la lista (index), por otro (element).
void	**add(int index, E element)**	Se añade otra versión del método add, en la cual se puede insertar un elemento (element) en la lista en una posición concreta (index), desplazando los existentes.
E	**remove(int index)**	Se añade otra versión del método remove, esta versión permite eliminar un elemento indicando su posición en la lista.
boolean	**addAll(int index, Collection<? extends E> c)**	Se añade otra versión del método addAll, que permite insertar una colección pasada por parámetro en una posición de la lista, desplazando el resto de elementos.
int	**indexOf(Object o).**	El método indexOf permite conocer la posición (índice) de un elemento, si dicho elemento no está en la lista retornará -1.
int	**lastIndexOf(Object o).**	El método lastIndexOf nos permite obtener la última ocurrencia del objeto en la lista (dado que la lista si puede almacenar duplicados).
List<E>	**subList(int from, int to)**	El método subList genera una sublista (una vista parcial de la lista) con los elementos comprendidos entre la posición inicial (incluida) y la posición final (no incluida).

<E> corresponde con el tipo base usado como parámetro genérico al crear la lista.

--LinkedList--

Implementa 2 interfaces MÁS, permiten hacer usar las listas como si fueran una **cola** de prioridad o una **pila,** respectivamente.

INTERFACES QUE IMPLEMENTA	java.util.**Queue** / java.util.**Deque.**

-COLAS-

En las Colas el que **primero** llegar es el **primero** en ser **atendido** (FIFO)

TIPO	MÉTODO	DESCRIPCIÓN
boolean	**add(E e)** **offer(E e)**	retornarán true si se ha podido insertar el elemento al final de la LinkedList.
E	**poll()**	retornará el primer elemento de la LinkedList y lo eliminará de la misma. Al insertar al final, los elementos más antiguos siempre están al principio. Retornará null si la lista está vacía.
E	**peek().**	retornará el primer elemento de la LinkedList pero no lo eliminará, permite examinarlo. Retornará null si la lista está vacía

-PILAS-

En las Pilas el **último** en llegar es el **primero** en ser **atendido**.

TIPO	MÉTODO	DESCRIPCIÓN
	push	meter al principio de la pila
	pop	sacar y eliminar del principio de la pila
	peek	y examinar el primer elemento de la pila

++ITERADORES (List)++

Necesitamos ITERADORES para acceder a los elementos almacenados en las listas. (podemos usar for-each o Iterator)

+ ITERADOR FOR-EACH +

→ (Utilizar SOLO Para LEER) (NO BORRAR ELEMENTOS CON ESTE BUCLE)

Es el más sencillo de implementar (recomendable)

Bucle **FOR-EACH**	**for (Integer i: ListaArrayList) {** System.out.println("Elemento almacenado:"+i); **}**

+"ITERATOR", (Tradicional)+

→ (UTILIZAR ESTE ITERADOR Y SU MÉTODO "REMOVE" EN CASO DE QUERER REMOVER ELEMENTOS)

Nos proporciona 3 MÉTODOS que debemos utilizar para poder iterar la Lista

TIPO	METODO	DESCRIPCIÓN
boolean	**hasNext()**	Retornará true si le quedan más elementos a la colección por visitar. False en caso contrario.
E	**next()**	Retornará el siguiente elemento de la colección, si no existe siguiente elemento, lanzará una excepción (NoSuchElementException), con lo que conviene chequear primero si el siguiente elemento existe.
	remove()	Elimina de la colección el último elemento retornado en la última invocación de next (no es necesario pasarselo por parámetro). **Cuidado, si next no ha sido invocado** todavía, saltará una incomoda excepción.

CREANDO ITERATOR (Tradicional)

***ITERADOR** **+** **Bucle WHILE** **+** **Métodos del Iterador**	**Iterator<**OBJ**> it=**ListArrayList.**iterator();** *//Creamos el iterador de el Objeto a Iterar.* **while** (it.**hasNext**()) **{** *// Mientras que haya un siguiente elemento, seguiremos en el bucle.* OBJ ListArrayList=it.**next**(); *// Escogemos el siguiente elemento.* if (ListArrayList%2==0) it.**remove(); }** *//Podemos eliminar el elemento extraído de la lista.*

Operaciones ADICIONALES sobre LISTAS y ARRAYS

NO aplicable a SET(conjuntos), ni a MAP(mapas)

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);**

MÉTODO "SPLIT"

Divide una cadena en Partes. Mientras estén delimitados por un separador (una coma, un punto y coma o cualquier otro)

String texto="**Z,B,A,X,M,O,P,U";** **String[]** partes=texto.**split(",");** ***Arrays.sort(partes);***	El delimitador o separador es una expresión regular, único argumento del método split, y puede ser obviamente todo lo complejo que sea necesario: Lo almacenará en un ARRAY,. (En el ejemplo, lo ordena)