Métodos de la interface `Collection`

`Collection` es una interfaz genérica donde "<E>" es el parámetro de tipo (podría ser cualquier clase):

método	uso
boolean add(E element)	Añade el elemento recibido a la colección. Devuelve **true** si se ha podido realizar la operación. Si no se ha podido realizar porque la colección no permite duplicados devuelve **false.**
boolean remove(E element)	Elimina el elemento indicado de la colección y devuelve **true**. Si no lo borra porque no existe devuelve **false**.
int size()	Devuelve el número de objetos almacenados en la colección.
boolean isEmpty()	Devuelve verdadero si la colección está vacía.
boolean contains(E element)	Devuelve **true** si la colección contiene al objeto indicado ***element***.
void clear()	Elimina todos los elementos de la colección.
boolean addAll(Collection<? Extends E> otra**)**	Permite añadir todos los elementos de la colección ***otra*** a la colección actual siempre que sean del mismo tipo (o deriven del mismo tipo base).
boolean removeAll(Collection<?> otra**)**	Si los elementos de la colección pasada como parámetro están en nuestra colección, se eliminan, el resto se quedan.
boolean retainAll(Collection<?> otra**)**	Si los elementos de la colección pasada como parámetro están en nuestra colección, se dejan, el resto se eliminan.
boolean containsAll(Collection<?> otra**)**	Si la colección contiene todos los elementos de otra colección devuelve verdadero.
Object[] toArray()	Permite pasar la colección a un array de objetos tipo Object.
<T> T[] toArray(T[] array)	Convierte la colección en un array de objetos. El array devuelto contiene todos los elementos de la colección y es del mismo tipo que el array que recibe como argumento (de hecho es la única utilidad que tiene este argumento, la de decir el tipo de array que se ha de devolver).
Iterator<E> iterator()	Crea un objeto iterador para recorrer los elementos de la colección.

Métodos de la interface `Iterator`

Una vez añadidos los datos a una colección, si queremos recorrerlos tenemos que usar un objeto de tipo **`Iteraror`**. Este objeto sirve como una "referencia" que se va moviendo por la colección.

método	uso
E next()	Hace que el iterador apunte al siguiente objeto de la colección. Si no hay más elementos lanza una excepción `NoSuchElementException` (que deriva a su vez de `RunTimeException`).
boolean hashNext()	Devuelve `true` si hay otro elemento después del objeto al que apunta el iterador y `false` en caso contrario. Este método **no modifica** el valor del iterador.
void remove()	Elimina el elemento al que apunta el iterador, que es el último elemento devuelto por **`next()`**.

Métodos de la interface `List`

método	uso
void add(int índice, E elemento)	Añade el elemento indicado en la posición `índice` de la lista.
E remove(int índice)	Elimina el elemento cuya posición en la colección la da el parámetro `índice`. Devuelve el elemento borrado.
E set(int índice, E elemento)	Sustituye el elemento que está en la posición `índice` por el que recibe como parámetro. Devuelve además el elemento antiguo
E get(int índice)	Obtiene el elemento almacenado en la colección en la posición que indica el `índice`.
int indexOf(Object elemento)	Devuelve la posición de la primera ocurrencia del elemento especificado. Si no lo encuentra, devuelve -1.
int lastIndexOf(Object elemento)	Devuelve la posición de la última ocurrencia del elemento especificado. Si no lo encuentra, devuelve -1
boolean addAll(int índice, Collection<? extends E> c)	Inserta todos los elementos de la colección c en el objeto de tipo `List` desde el que se ha invocado el método. El elemento que estaba en la posición índice y todos los que estaban a su derecha se desplazan hacia la derecha (incrementan sus índices).
ListIterator<E> listIterator( )	Devuelve un iterador al principio de la lista desde la que se ha invocado al método.
ListIterator<E> listIterator(int índice)	Devuelve un iterador al elemento que se encuentra en el `índice` especificado. Si el índice está fuera de rango lanza una excepción `IndexOutOfBoundsException`.
List<E> subList(int principio, int final)	Devuelve una lista que incluye los elementos que ocupan los índices desde `principio` hasta `final`.

Cualquier error en los índices produce `IndexOutOfBoundsException`

Métodos de la interface `ListIterator`

`ListIterator` hereda de `Iterator` y añade métodos para permitir el recorrido bidireccional de una lista.

método	uso
void add(E objeto)	Inserta en la colección el objeto que recibe. Lo inserta a continuación del objeto al que apunta el iterador.
void set(E objeto)	Sustituye el elemento señalado por el iterador, por el elemento que recibe como parámetro.
E previous()	Obtiene el elemento previo al actual. Si no lo hay provoca excepción: `NoSuchElementException`.
boolean hasPrevious()	Devuelve true si hay un elemento anterior al actualmente señalado por el iterador.
int nextIndex()	Devuelve el índice del elemento siguiente al que apunta el iterador. Si no hay un elemento a continuación, devuelve el tamaño de la lista.
int previousIndex()	Obtiene el índice del elemento anterior al que apunta el iterador. Si no hay un elemento previo devuelve -1.

Clase `ArrayList`

Implementa la interfaz `List` y es la clase fundamental para representar colecciones de datos

Tiene tres constructores:

ArrayList(). Constructor por defecto. Simplemente crea un ArrayList vacío

ArrayList(int capacidadInicial**).** Crea una lista con una capacidad inicial indicada.

ArrayList(Collection<? extends E> ***c*****).** Crea una lista a partir de los elementos de la colección indicada.

Clase `LinkedList`

Hereda de **`AbstractSequentialList`** e implementa las interfaces **`List`, `Deque`** y **`Queue`**. Desde esta clase es sencillo implantar estructuras en forma de pila, cola o lista doblemente enlazada. **`LinkedList`** es una clase generica que se declara:

```
class LinkedList<E>
```

Donde **E** especifica el tipo de objetos que tendrá la lista. **LinkedList** tiene dos constructores:

- **`LinkedList( ):`** Crea una lista enlazada vacía.
- **`LinkedList(Collection<? extends E> c) ):`** Crea una lista enlazada que contiene los elementos de la colección c.

Añade los métodos:

método	uso
Object getFirst()	Obtiene el primer elemento de la lista
Object getLast()	Obtiene el último elemento de la lista
void addFirst(Object o**)**	Añade el objeto al principio de la lista
void addLast(Object o**)**	Añade el objeto al final de la lista
void removeFirst()	Borra el primer elemento
void removeLast()	Borra el último elemento

Clase `HashSet`

Esta clase hereda de **`AbstractSet`** e implementa la interface **`Set`**, que a su vez hereda de **`Collection`**. Es una clase genérica que se declara:

```
Class HashSet<E>
```

Los objetos HashSet se construyen con un tamaño inicial de tabla (el tamaño del array) y un factor de carga que indica cuándo se debe redimensionar el array. Es decir si se creó un array de 100 elementos y la carga se estableció al 80%, entonces cuando se hayan rellenado 80 valores únicos, se redimensiona el array.

Por defecto el tamaño del array se toma con 16 y el factor de carga con 0,75 (75%). No obstante se puede construir una lista **`HashSet`** indicando ambos parámetros. Los posibles constructores de la clase **`HashSet <E>`** son:

Constructor	uso
HashSet()	Construye una nueva lista vacía con tamaño inicial 16 y un factor de carga de 0,75.
HashSet(Collection<? extends E> c)	Crea una lista Set a partir de la colección compatible indicada.
HashSet(int *capacity*)	Crea una lista con el tamaño indicado y un factor de 0,16.
HashSet(int *capacity*, float *fillRatio*)	Crea una lista con la capacidad y el factor indicados.

Clase `TreeSet`

Hereda de **`AbstractSet`** e implementa, entre otras, las interface **`Set`** y **`SortedSet`** y las clases de los elementos almacenados deben implementar la interface `Comparable`, o bien durante la creación de un objeto **`TreeSet`** pasar como parámetro al constructor un objeto **`Comparator`**. Es una clase genérica que se declara:

```
Class TreeSet<E>
```

Tiene los siguientes constructores:

Constructor	uso
TreeSet()	Construye un árbol vacío que se ordenará en orden ascendente de acuerdo al orden natural de sus elementos.
TreeSet (Collection<? extends E> c)	Crea un árbol que contiene los elementos de la colección c.
TreeSet (Comparator<?super E> comp)	Crea un árbol vacío cuyos elementos se ordenarán de acuerdo al objeto Comparator especificado (*).
TreeSet (SortedSet<E> ss)	Crea una lista con la capacidad y el factor indicados.

Por defecto esta clase ordena sus elementos por lo que se puede considerar el "orden natural": "A", "B", "C",…, 1, 2, 3,… Si queremos ordenar los elementos de otra forma tenemos que especificar un `Comparator` en el constructor. En la clase que utilicemos para crear dicho objeto habrá que implementar los métodos **`equals()`** y **`compare()`**.

Interface `Map`

Los métodos principales de la interfaz **`Map`**, disponibles en todas las implementaciones. En los ejemplos, `V` es el tipo base usado para el valor y `K` el tipo base usado para la llave:

método	uso
V get(K clave**)**	Devuelve el valor asociado a la clave indicada. Si no existe esa clave devuelve null.
V put(k clave, **V** valor**)**	Coloca el par clave-valor en el mapa (asociando la clave a dicho valor). Si la clave ya existiera, sobrescribe el anterior valor y devuelve el valor antiguo. Si esa clave no aparecía en la lista, devuelve **null**
V remove(k clave**)**	Elimina de la lista el nodo asociado a esa clave. Devuelve el valor que tuviera asociado esa clave o **null** si esa clave no existe en el mapa.
boolean containsKey(Object clave**)**	Indica si el mapa posee la clave señalada
boolean containsValue(Object valor**)**	Indica si el mapa posee el valor señalado
void putAll(Map<? extends K, ? extends V> mapa**)**	Añade todo el mapa indicado, al mapa actual
Set<K> keySet()	Obtiene un objeto *Set* creado a partir de las claves del mapa
Collection<V> values()	Obtiene la colección de valores del mapa, permite utilizar el **HashMap** como si fuera una lista normal al estilo de las que implementan la interface **Collection** (por lo tanto se permite recorrer cada elemento de la lista con un iterador)

int size()	Devuelve el número de pares clave-valor del mapa
Set<Map.Entry <K,V>> entrySet()	Devuelve una lista (Set) formada por objetos **Map.Entry**. Esto permite obtener una "vista como colección" del mapa en el que se invoca.
void clear()	Elimina todos los objetos del mapa

Interface `Map.Entry<K,V>`

La interfaz **`Map.Entry`** se define de forma interna a la interfaz **`Map`** y representa un objeto de par clave/valor. Es decir mediante esta interfaz podemos trabajar con una entrada del mapa. Tiene estos métodos:

método	uso
K getKey()	Obtiene la clave del elemento actual **Map.Entry.**
V getValue()	Obtiene el valor.
V setValue(V valor**)**	Cambia el valor del elemento actual y devuelve el valor que tenía antes de hacer el cambio.
boolean equals(Object obj**)**	Devuelve verdadero si el objeto es un **Map.Entry** cuyos pares clave-valor son iguales que los del **Map.Entry** actual.

Clase `HashMap`

método	uso
HashMap()	Crea un mapa con una tabla Hash de tamaño 16 y factor de carga de 0,75.
HashMap(Map<?extends K, ?extends V> m**)**	Crea un mapa colocando los elementos de otro.
HashMap(int capacidad**)**	Crea un mapa con la capacidad indicada y factor de carga de 0,75.
HashMap(int capacidad, **float** factorCarga**)**	Crea un mapa con la capacidad y factor de carga máximo indicado.

Clase `TreeMap`

método	uso
TreeMap()	Crea un mapa vacío que utiliza el orden natural establecido en las claves.
TreeMap(Comparator<? super K> comp**)**	Crea un mapa vacío que se ordenará usando el criterio del objeto `Comparator` comp.
TreeMap(Map<?extends K, ?extends V> m**)**	Crea un mapa a partir de los elementos del mapa m, que se ordenarán según el orden de las claves que se haya definido en el nuevo.
TreeMap(SortedSet<K, ?extends V> m**)**	Crea un mapa usando los elementos y orden de otro mapa.

Clase `LinkedHashMap`

método	uso
LinkedHashMap()	Crea un mapa con una tabla Hash de tamaño 16 y factor de carga de 0,75.
LinkedHashMap(**Map<?extends K, ?extends V>** m**)**	Crea un mapa colocando los elementos de otro.
LinkedHashMap(int capacidad**)**	Crea un mapa con la capacidad indicada y factor de carga de 0,75.
LinkedHashMap(int capacidad, **float** factorCarga**)**	Crea un mapa con la capacidad y factor de carga máximo indicado.
LinkedHashMap(int capacidad, **float** factorCarga, **boolean** orden**)**	Crea un mapa con la capacidad y factor de carga máximo indicado. Además si el último parámetro (***orden***) es verdadero el orden se realiza según el orden del último acceso y si es falso el orden es por orden de inserción.