Tema 7. Colecciones en Java

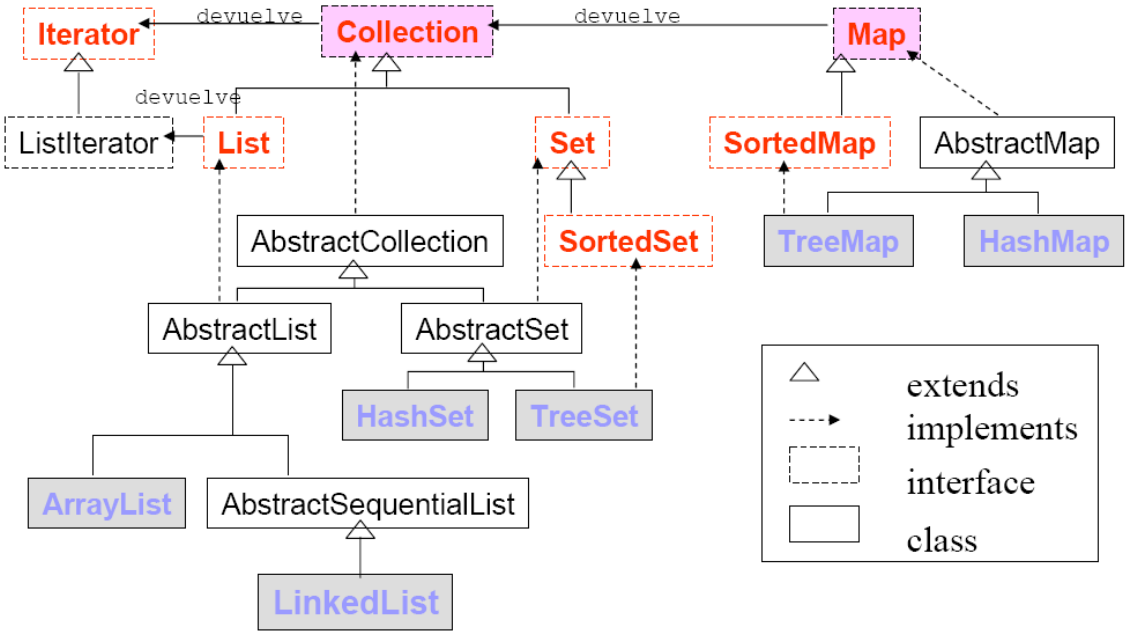
1 Introducción a las colecciones.

Para ver información más detallada consultar:

http://docs.oracle.com/javase/tutorial/collections/TOC.html

Una **colección** es un grupo de objetos almacenados de forma conjunta en una misma estructura. A estos objetos se les llama elementos y para poder trabajar con ellos utilizaremos la **interface** genérica **`Collection`**. Gracias a esta interface, 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… De la interface genérica **`Collection`** heredan otra serie de interfaces genéricas. Estas subinterfaces imponen más restricciones y aportan distintas funcionalidades sobre la interface anterior.

La jerarquía de interfaces y clases que permiten manipular este tipo de estructuras es la siguiente:



Para **obtener los elementos** almacenados **en una colección** hay que **usar iteradores**, que permiten obtenerlos uno a uno de forma secuencial (*no hay otra forma de acceder a los elementos de una colección*). Los iteradores, que veremos más adelante, se pueden usar de forma transparente, a través de una estructura for especial, denominada bucle "`for-each`" o bucle "para cada". En el siguiente código se usa un bucle `for-each`, en él la variable ***i*** va tomando todos los valores almacenados en el conjunto hasta que llega al último:

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

La estructura for-each es muy sencilla: la palabra for seguida de "**`(tipo variable:colección)`**" y el cuerpo del bucle; *`tipo`* es el tipo del objeto sobre el que se ha creado la colección, *`variable`* es la variable donde se almacenará cada elemento de la colección y *`colección`* es la colección en sí. Los bucles `for-each` se pueden usar para todas las colecciones.

EjemploHashSet

2 Tipos de colecciones

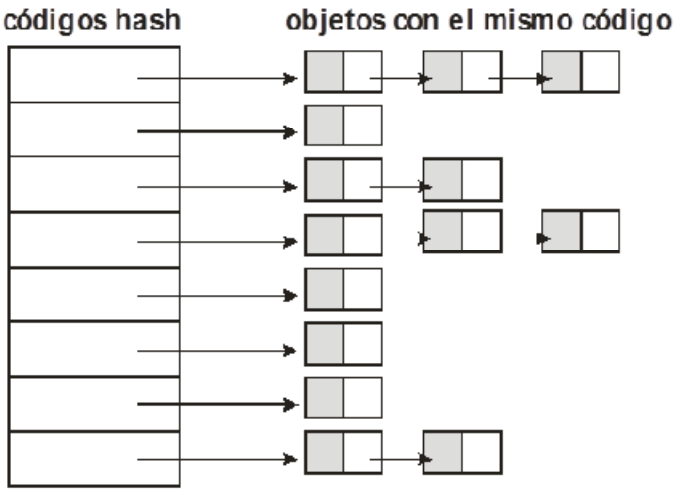
Hay varias interfaces que **heredan** de la interface **`Collection`**, y por ahora vamos a ver las interfaces **`Set`** y **`List`**:

2.1 Interface Set

La **interface `Set`** define una colección que **no puede contener** elementos **duplicados**. Esta interface contiene, únicamente, los métodos heredados de **`Collection`** añadiendo la restricción de que los elementos duplicados están prohibidos. Para comprobar si los elementos están duplicados o no, es necesario que las clases de los objetos que se almacenan en la colección implementen de forma correcta los métodos **`equals`** y **`hashCode`**. Para comprobar si dos **`Set`** son iguales, se comprobará si todos los elementos que los componen son iguales sin importar el orden que ocupen dichos elementos.

La **interface `Set`** es implementada, entre otras por las **clases**:

- **`HashSet (java.util.HashSet)`**: Es la clase más utilizada para implementar una colección sin duplicados. Almacena los elementos en una tabla *hash* (tabla que permite asociar clave con valores). La igualdad de los objetos se comprueba comparando los códigos hash y, si son iguales, se compara con `equals`. Es la implementación con mejor rendimiento de todas pero **no almacena** los objetos **de forma ordenada**.



- De la clase anterior hereda **`LinkedHashSet (java.util.LinkedHashSet)`**, que almacena los objetos combinando tablas hash, para un acceso rápido a los datos, y listas enlazadas para recuperarlos **en el orden en que fueron insertados**.

- **`TreeSet (java.util.TreeSet)`**: esta clase implementa la interface **`SortedSet`**, que almacena **los objetos usando** unas estructuras conocidas como **árboles** rojo-negro, **que los ordenan en función de sus valores**, de forma que aunque se inserten los elementos de forma desordenada, internamente se ordenan dependiendo del valor de cada uno. Es bastante más lento que **`HashSet`**. Las clases de los elementos almacenados deben implementar la interface **`Comparable`** de Java (está en **`java.lang`**). Esta interface define el método **`compareTo`** que utiliza como argumento un objeto a comparar y que devuelve 0 si los objetos son iguales, un número positivo si el primero es mayor que el segundo y negativo en caso contrario.

2.2 Interface List

La **interface `List`** define una sucesión de elementos que están dispuestos en un cierto orden. A diferencia de la interface **`Set`**, la interface **`List`** sí **admite** elementos **duplicados**. Aparte de los métodos heredados de **`Collection`**, añade métodos que permiten mejorar los siguientes puntos:

- Acceso posicional a elementos: manipula elementos en función de su posición en la lista. Las posiciones se empiezan a numerar desde cero.
- Búsqueda de elementos: busca un elemento concreto de la lista y devuelve su posición.
- Rango de operación: permite realizar ciertas operaciones sobre rangos de elementos dentro de la propia lista, por ejemplo insertar los elementos de otra colección a partir de una posición dada.

La **interface `List`** es implementada por dos **clases**:

- **`ArrayList (java.util.ArrayList)`**: esta es la implementación típica. Se basa en un *array* redimensionable que aumenta su tamaño según crece la colección de elementos. La redimensión es transparente al usuario, nosotros, no nos enteramos cuando se produce, pero eso redunda en una diferencia de rendimiento notable dependiendo del uso. Los ArrayList son muy rápidos para acceder a un elemento según su posición. En cambio, eliminar o insertar un elemento implica muchas operaciones ya que hay que mover todos los elementos que van después del nodo que queremos borrar o insertar.
- **`LinkedList (java.util.LinkedList)`**: esta implementación permite que mejore el rendimiento en ciertas ocasiones. Se basa en una lista doblemente enlazada de los elementos, teniendo cada uno de los elementos un puntero al anterior y al siguiente elemento.

  Esta clase también puede implementar las interfaces **`java.util.Queue`** y **`java.util.Deque`**. Dichas interfaces permiten hacer uso de las listas como si fueran una cola de prioridad o una pila, respectivamente.

Si se van a realizar muchas operaciones de eliminación de elementos sobre la lista, conviene usar una lista enlazada (**`LinkedList`**), pero si no se van a realizar muchas eliminaciones, sino que solamente se van a insertar y consultar elementos por posición, conviene usar una lista basada en arrays redimensionados (**`ArrayList`**).

2.3 Interface Map

Los mapas permiten definir colecciones de elementos que poseen pares de datos clave-valor, no pueden contener claves duplicadas y cada una de dichas claves, sólo puede tener asociado un valor como máximo. Esto se utiliza para localizar valores en función de la clave que poseen.

Es la raíz de todas las clases capaces de implementar mapas. Hasta la versión 1.5, los mapas eran colecciones de pares clave, valor donde tanto la clave como el valor eran de tipo **Object**. Desde la versión 1.5 esta interfaz tiene dos genéricos: **K** para el tipo de datos de la clave y **V** para el tipo de los valores (**`Map<k, V>`**).

Esta interfaz no deriva de **`Collection`** y no usa iteradores porque la obtención, búsqueda y borrado de elementos se hace de manera muy distinta, pero veremos que existe un truco interesante. Los mapas **no permiten insertar objetos nulos** (provocan excepciones de tipo **`NullPointerException`**).

Dentro de la interface **Map** existen varios tipos de implementaciones realizadas dentro de la plataforma Java. Vamos a analizar cada una de ellas:

- **HashMap (java.util.HashMap)**: 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. Esta implementación proporciona tiempos constantes en las operaciones básicas siempre y cuando la función *hash* disperse de forma correcta los elementos dentro de la tabla *hash*. Es importante definir el tamaño inicial de la tabla ya que este tamaño marcará el rendimiento de esta implementación.
- **TreeMap (java.util.TreeMap)**: esta implementación **almacena las claves ordenándolas** en función de sus valores. Es bastante más lento que *HashMap*. La clase de las claves debe implementar la interface **Comparable** o bien durante la creación de un objeto **TreeMap** pasar como parámetro al constructor un objeto **Comparator.**
- **LinkedHashMap (java.util.LinkedHashMap)**: esta implementación almacena las claves en función del **orden de inserción**. Es, simplemente, un poco más costosa que **HashMap**.

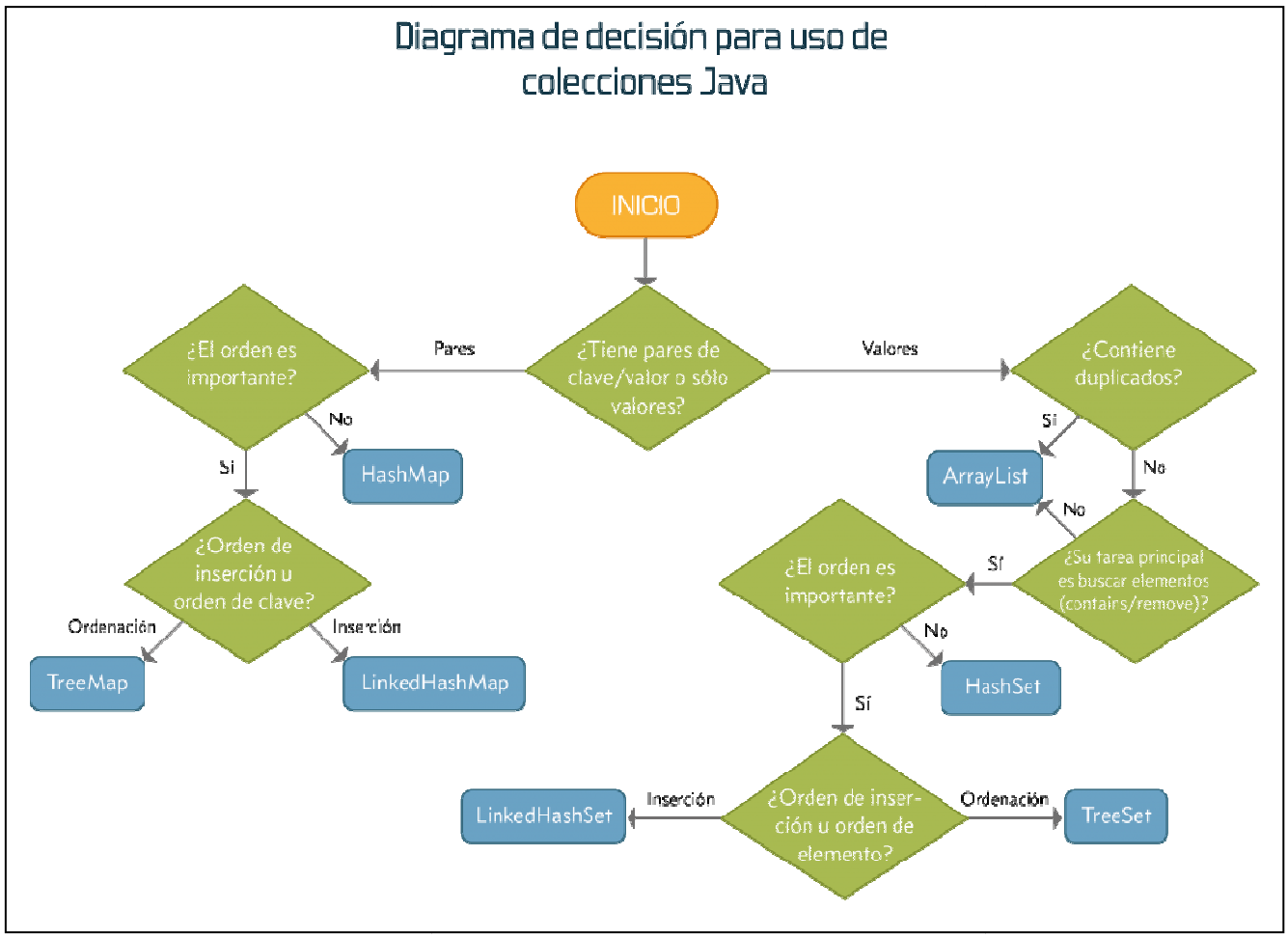
Los **mapas utilizan clases genéricas** y permiten definir un tipo base para la clave, y otro tipo diferente para el valor. Veamos un ejemplo de cómo crear un mapa, que es extensible a los otros dos tipos de mapas:

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

El mapa anterior permite usar cadenas como claves y almacenar de forma asociada a cada clave, un número entero.

El cuándo usar una implementación u otra de **Map** variará en función de la situación en la que nos encontremos. Generalmente, **HashMap** será la implementación que usemos en la mayoría de situaciones. **HashMap** es la implementación con mejor rendimiento, pero en algunas ocasiones podemos decidir renunciar a este rendimiento a favor de cierta funcionalidad como la ordenación de sus elementos.

Para conocer qué tipo de colección usar, podemos emplear el siguiente diagrama:



2.4 Interface Iterator (java.util.Iterator)

La interface `Collection` tiene el método **`iterator()`** que devuelve un iterador, que es un objeto que implementa, o bien la interface `Iterator`, o bien la interface `ListIterator`, y con los métodos de este objeto podemos recorrer la colección.

Las colecciones se pueden recorrer de dos formas: con **bucles for-each** (existentes en Java a partir de la versión 1.5) **y a través de un bucle normal creando un iterador**. Los bucles for-each ya los hemos visto. Nos falta ver cómo se crea un iterador: sólo hay que invocar al método "`iterator()`" de cualquier colección.

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

Hemos especificado un parámetro para el tipo de dato genérico en el iterador (poniendo "`<Integer>`" después de `Iterator`) porque es una interface genérica y es necesario especificar el tipo base que contendrá el iterador. Si no se especifica el tipo base del iterador, igual se puede recorrer la colección, pero devolverá objetos tipo `Object` (clase de la que derivan todas las clases), con lo que nos veremos obligados a forzar la conversión de tipo.

Para recorrer y gestionar la colección, el iterador ofrece tres métodos básicos:

- `boolean hasNext()`. Devuelve *true* si le quedan más elementos por visitar en la colección y *False* en caso contrario.
- `E next()`. Devuelve el siguiente elemento de la colección, si no existe siguiente elemento lanzará la excepción `NoSuchElementException`, por eso conviene chequear primero si el siguiente elemento existe.

- `remove()`. Elimina de la colección el último elemento devuelto en la última invocación de *next* (no es necesario pasárselo como parámetro). Si *next* no ha sido invocado todavía saltará una excepción.

EjemploIterator01

2.5 Interface ListIterator

Hereda de la interface `Iterator`, por tanto puede usar sus métodos y además añade otros nuevos para modificar elementos o recorrer la lista en cualquier sentido.

3 Recorrido de mapas

Un iterador está pensado para no sobrepasar los límites de la colección, ocultando operaciones más complicadas que pueden producir errores. Puede haber problemas cuando es necesario hacer la operación de conversión de tipos. Si la colección no contiene los objetos esperados, al intentar hacer la conversión, saltará una excepción. Usar genéricos aporta grandes ventajas, pero hay que usarlos adecuadamente.

Para recorrer los mapas con iteradores, hay que hacer un pequeño truco. Usamos el método **`entrySet`** que ofrecen los mapas para generar un conjunto con las entradas (pares de llave-valor), o bien, el método **`keySet`** para generar un conjunto con las claves existentes en el mapa. Veamos cómo sería para el segundo caso, con **`keySet`**, que es el más sencillo:

```
Map<Integer,Integer> mapa=new HashMap<Integer,Integer>();

for (int i=1;i<=10;i++) mapa.put(i, i*2);//Insertamos datos de prueba en el mapa.

for (Integer clave:mapa.keySet()) // Recorremos el conjunto generado por keySet,
                                  // contendrá las claves.
{
   Integer valor=mapa.get(clave); //Para cada clave, accedemos a su valor si
                                  // es necesario.
   System.out.println(valor);
}
```

Lo único que hay que tener en cuenta es que el conjunto generado por `keySet` no tendrá el método `add` para añadir elementos al mismo, dado que eso habrá que hacerlo a través del mapa.

Si usas iteradores, y piensas eliminar elementos de la colección (e incluso de un mapa), debes usar el método remove del iterador y no el de la colección. Si eliminas los elementos utilizando el método remove de la colección, mientras estás dentro de un bucle de iteración los fallos que pueden producirse en tu programa son impredecibles.

Los problemas son debidos a que el método remove del iterador elimina el elemento de dos sitios: de la colección y del iterador en sí (que mantiene interiormente información del orden de los elementos). Si usamos el método remove de la colección, la información solo se elimina de un lugar, de la colección.

CUADRO RESUMEN DE COLECCIONES

	Repetición de clave	No hay repetición de clave	Estar ordenado por clave	Ordenado orden de inserción	Par clave-valor	Solo clave
ArrayList	X	X				X
LinkedList	X	X				X
TreeSet		X	X			X
LinkedHashSet (Extend HashSet)		X		X		X
HashSet		X		X		X
TreeMap		X	X		X	X
HashMap		X		X	X	X
LinkedHashMap		X		X	X	X

Métodos definidos para cada Interfaz:

	List	**Set**	**Map**	**devuelve**
Colección vacía	.isEmpty()	.isEmpty()	.isEmpty()	*boolean*
Tamaño	.size()	.size()	.size()	*int*
Borrar colección	.clear()	.clear()	.clear()	*void*
Añadir	.add(E e) .add(int pos, E e)	.add(E e)	.put(K key, V value)	*boolean/V*
Eliminar por clave	-	-	.remove(Object key)	*V*
Eliminar	.remove(Object o)	.remove(Object o)	.remove(Object key, Object value)	*boolean*
Buscar clave	.contains(Object o)	.contains(Object o)	.containsKey(Object key)	*boolean*
Buscar valor	-	-	.containsValue(Object value)	*boolean*
Modificar	.set(int index, E element)*	-	.replace(K key, V oldValue, V newValue)	*E/boolean*
Obtener posición	indexOf(Object o)*	-	-	*int*
Obtener valor	.get(int index)*		.get(Object key)	*E/V*
Convertir en set			.entrySet()	*Set<Map.Entry<K,V>*
Convertir en set claves	-	-	.keySet()	*Set< K>*
Convertir en set valores	-	-	.values()	*Collection<V>*
Iterar	.iterator()	.iterator()		*Iterator<E>*
Iterar inverso	.descendingIterator()	.descendingIterator()		*Iterator<E>*
Ordenar	.sort(Comparator<? Super E c>	-	-	*void*
Asociar	-	-	.put(K key, V value)	*V*
Comparar		.comparator()	.comparator()	*Comparator<? Super E/K>*

Métodos de la interface `Collection`

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

Devuelve	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.

Devuelve	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`

Devuelve	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.

Devuelve	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:

Constructor	uso
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:

Devuelve	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
Object	**.removeFirst()**	Borra el primer elemento
Object	**.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:

Devuelve	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 la clase **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:

Devuelve	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`

Constructor	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`

Constructor	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`

Constructor	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.