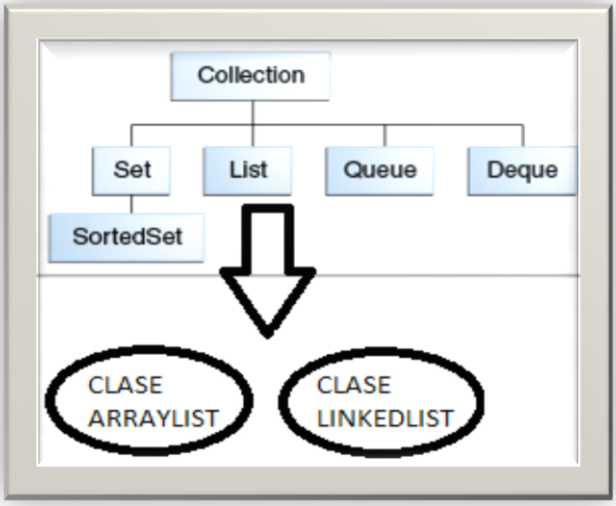
Tema 5: Estructuras de datos.



- Set: Es una colección que no puede tener elementos duplicados.
- List: Es una colección **ordenada**.

Puede contener elementos duplicados. Accede a los elementos mediante un índice entero.

- Colas (Queue y Deque) contienen elementos con prioridad para ser procesados.

Nos vamos a centrar en la interfaz LIST. La implementan dos clases: ARRAYLIST y LINKEDLIST

1.1 DIFERENCIAS ENTRE Set y List

Las listas (List) son elementos de programación un poco más avanzados que los conjuntos (Set). Su ventaja es que amplían la relación de operaciones de las colecciones añadiendo operaciones extra, por ejemplo:

- Las listas si pueden almacenar duplicados, si no queremos duplicados, hay que verificar manualmente que el elemento no esté en la lista antes de su inserción.
- Acceso posicional. Podemos acceder a un elemento indicando su posición en la lista.
- Búsqueda. Es posible buscar elementos en la lista y obtener su posición. En los conjuntos, al ser colecciones sin aportar nada nuevo, solo se puede comprobar si un conjunto contiene o no un elemento de la lista, retornando verdadero o falso. Las listas mejoran este aspecto.
- Extracción de sublistas. Es posible obtener una lista que contenga solo una parte de los elementos de forma muy sencilla.

En Java, para las listas se dispone de una interfaz llamada **java.util.List**, y dos implementaciones (**java.util.LinkedList** y **java.util.ArrayList**), con diferencias significativas entre ellas. Los métodos de la interfaz List, que obviamente estarán en todas las implementaciones, y que permiten las operaciones anteriores son:

- 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 list .

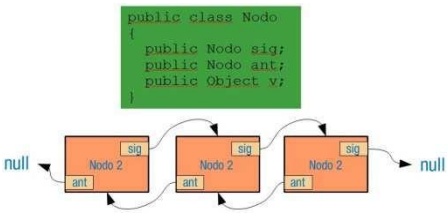
- boolean addAll(int index, Collection<? extends E> c). Se añade otra versión del método addAll, que permite insertar una colección pasada por parámetro en una posición de la lista, desplazando el resto de elementos.
- int indexOf(Object o). El método indexOf permite conocer la posición (índice) de un 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).

Ten en cuenta que los elementos de una lista empiezan a numerarse por 0. Es decir, que el primer elemento de la lista es el 0. Ten en cuenta también que **List** es una interfaz genérica, por lo que **<E>** corresponde con el tipo base usado como parámetro genérico al crear la lista.

1.2 LinkedList y ArrayList

Para usar una lista haremos uso de sus implementaciones **LinkedList** y **ArrayList**.

Los **LinkedList** utilizan listas doblemente enlazadas, que son listas enlazadas, pero que permiten ir hacia atrás en la lista de elementos.



Los elementos de la lista se encapsulan en los llamados nodos. Los nodos van enlazados unos a otros para no perder el orden y no limitar el tamaño de almacenamiento. Tener un doble enlace significa que en cada nodo se almacena la información de cuál es el siguiente nodo y además, de cuál es el nodo anterior. Si un nodo no tiene nodo siguiente o nodo anterior, se almacena null o nulo para ambos casos.

No es el caso de los **ArrayList**. Estos se implementan utilizando arrays que se van redimensionando conforme se necesita más espacio o menos. La redimensión es transparente a nosotros, no nos enteramos cuando se produce, pero eso redunda en una diferencia de rendimiento notable dependiendo del uso. Los **ArrayList** son más rápidos en cuanto a acceso a los elementos, acceder a un elemento según su posición es más rápido en un array que en una lista doblemente enlazada (hay que recorrer la lista). En cambio, eliminar un elemento implica muchas más operaciones en un array que en una lista enlazada de cualquier tipo.

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).

LinkedList tiene otras ventajas que nos puede llevar a su uso. Implementa 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.

Las colas, también conocidas como colas de prioridad, son una lista que trata de que el que primero llega es el primero en ser atendido (FIFO). Simplemente se aportan tres métodos nuevos: meter en el final de la lista (**add** y **offer**), sacar y eliminar el elemento más antiguo (**poll**), y examinar el elemento al principio de la lista sin eliminarlo (**peek**). Dichos métodos están disponibles en las listas enlazadas **LinkedList**:

- boolean add(E e) y boolean 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.

Las pilas, mucho menos usadas, son todo lo contrario a las colas; en las pilas el último en llegar es el primero en ser atendido. Para ello se proveen de tres métodos: meter al principio de la pila (**push**), sacar y eliminar del principio de la pila (**pop**), y examinar el primer elemento de la pila (**peek**, igual que si usara la lista como una cola).

1.3 Mapas

¿Cómo almacenar los datos de un diccionario? Tenemos por un lado cada palabra y por otro su significado. Para resolver este problema existen precisamente los arrays asociativos. Un tipo de array asociativo son los mapas o diccionarios, que permiten almacenar pares de valores conocidos como clave y valor. La clave se utiliza para acceder al valor, como una entrada de un diccionario permite acceder a su definición.

En Java existe la interfaz **java.util.Map** que define los métodos que deben tener los mapas, y existen tres implementaciones principales de dicha interfaz: **java.util.HashMap**, **java.util.TreeMap** y **java.util.LinkedHashMap**.

Los **mapas utilizan clases genéricas** para dar extensibilidad y flexibilidad, 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 llaves y almacenar de forma asociada a cada llave, un número entero.

Veamos 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:

Métodos principales de los mapas.

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.

1.4 Iteradores

Los iteradores son un mecanismo que nos permite recorrer todos los elementos de una colección de forma sencilla, de forma secuencial, y de forma segura. Los mapas, como no derivan de la interfaz Collection realmente, no tienen iteradores formales, pero veremos cómo recorrerlos.

Los iteradores permiten recorrer las colecciones de dos formas:

- **Bucles for-each**

Es 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 consiste en la palabra **for** seguida de **"(Tipo variable : colección)**" y el cuerpo del bucle, donde

- **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.
- **Colección** es el identificador la colección en sí.

Los bucles for-each se pueden usar para todas las colecciones.

- **Creando un iterador**.

Para ello es necesario invocar el método "**iterator()**" de cualquier colección.

Veamos un ejemplo (en el ejemplo **t** es una colección cualquiera, por ejemplo con los enteros del 1 al 10):

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

Se ha especificado un parámetro para el tipo de dato genérico en el iterador (poniendo **"<Integer>**" después de **Iterator**). Esto es porque los iteradores son también clases genéricas, y es necesario especificar el tipo base que contendrá el iterador. Si no se especifica el tipo base del iterador, nos permitirá recorrer la colección, pero retornará 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(). 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 excepción.

Se recorrería una colección con estos métodos con un bucle (**while**) con la condición **hasNext()** que nos permite hacerlo:

```
while (it.hasNext()){ // Mientras que haya un siguiente elemento, seguiremos en el
                                bucle.
  Integer t=it.next(); // Escogemos el siguiente elemento.
  if  (t%2==0)  it.remove();    //Si es necesario, podemos eliminar el elemento
extraído de la lista.
}
```

¿Qué elementos contendría la lista después de ejecutar el bucle?

Observación

Las listas permiten acceso posicional a través de los métodos get y set, y acceso secuencial a través de iteradores. Se puede seleccionar por lo tanto el modo más cómodo para cada usuario de recorrer todos los elementos: un acceso posicional a través un bucle "for (i=0;i<lista.size();i++)" o un acceso secuencial usando un bucle "while (iterador.hasNext())"

Observación

Para recorrer los mapas con iteradores, hay que hacer lo siguiente. 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 llaves existentes en el mapa. Veamos cómo sería para el segundo caso, el más sencillo:

```
HashMap<Integer,Integer> mapa=new HashMap<Integer,Integer>test();
for (int i=0;i<10;i++) mapa.put(i, i); // Insertamos datos de prueba en el mapa.
for (Integer llave:mapa.keySet()) // Recorremos el conjunto generado por keySet,
contendrá las llaves.
{
        Integer valor=mapa.get(llave); //Para cada llave, accedemos a su valor si
es necesario.
}
```

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

Recomendación

Si se usan iteradores, y se piensa eliminar elementos de la colección (e incluso de un mapa), se debe 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, o dentro de un bucle for-each, 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 usas el método remove de la colección, la información solo se elimina de un lugar, de la colección.

1.5 Trabajando con arrays y con Listas

- Ordenar arrays y Listas.
- Desordenar arrays y Listas.
- Búsqueda binaria en arrays y Listas.
- Conversión de arrays a Listas y de Listas a array.
- Partir cadenas y almacenar el resultado en un array.

Estos algoritmos están en su mayoría recogidos como métodos estáticos de las clases **java.util.Arrays**, y **java.util.Collections**

Ordenación natural en arrays y Colecciones	
Ejemplo de ordenación de un array de números	**Ejemplo de ordenación de una Colección con números**

Integer[] array={10,9,99,3,5};
 	
Arrays.sort(array);
	ArrayList<Integer>lista
	=new ArrayList<Integer> ();
	lista.add(10); lista.add(9);lista.add(99);
	lista.add(3); lista.add(5);
	Collections.sort(lista);

Operaciones adicionales sobre listas y arrays.		
Operación	**Descripción**	**Ejemplos**
Desordenar una lista.	Desordena una lista, este método no está disponible para arrays.	Collections.shuffle (lista);
Rellenar una lista oarray.	Rellena una lista o array copiando el mismo valor en todos los elementos de larray 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 unarray del tipo almacenado en la lista, y del tamaño de la lista: Integer[] array=new Integer[lista.size()]; lista.toArray(array)