

# TEMA 3 COLECCIONES

## 1.- Introducción a las colecciones (I).

¿Qué consideras que es una colección? Pues seguramente al pensar en el término se te viene a la cabeza una colección de libros o algo parecido, y la idea no va muy desencaminada. **Una colección a nivel de software es un grupo de elementos almacenados de forma conjunta en una misma estructura.** Eso son las colecciones.

Las colecciones definen un conjunto de interfaces, clases genéricas y algoritmos que permiten manejar grupos de objetos, todo ello enfocado a potenciar la reusabilidad del software y facilitar las tareas de programación. Te parecerá increíble el tiempo que se ahorra empleando colecciones y cómo se reduce la complejidad del software usándolas adecuadamente. Las colecciones permiten almacenar y manipular grupos de objetos que, a priori, están relacionados entre sí (aunque no es obligatorio que estén relacionados, lo lógico es que si se almacenan juntos es porque tienen alguna relación entre sí), pudiendo trabajar con cualquier tipo de objeto (de ahí que se empleen los genéricos en las colecciones).

Además las colecciones permiten realizar algunas operaciones útiles sobre los elementos almacenados, tales como búsqueda u ordenación. En algunos casos es necesario que los objetos almacenados cumplan algunas condiciones (que implementen algunas interfaces), para poder ser hacer uso de estos algoritmos.

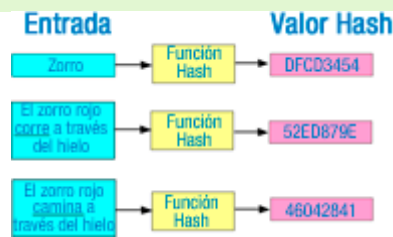
Las colecciones son en general elementos de programación que están disponibles en muchos lenguajes de programación. En algunos lenguajes de programación su uso es algo más complejo (como es el caso de C++), pero en Java su uso es bastante sencillo, es algo que descubrirás a lo largo de lo que queda de tema.

Las colecciones en Java parten de una serie de interfaces básicas. Cada interfaz define un modelo de colección y las operaciones que se pueden llevar a cabo sobre los datos almacenados, por lo que es necesario conocerlas. La interfaz inicial, a través de la cual se han construido el resto de colecciones, es la interfaz **java.util.Collection**, que define las operaciones comunes a todas las colecciones derivadas. A continuación se muestran las operaciones más importantes definidas por esta interfaz, ten en cuenta que **Collection** es una interfaz genérica donde "<E>" es el parámetro de tipo (podría ser cualquier clase):

- Método `int size()`: retorna el número de elementos de la colección.
- Método `boolean isEmpty()`: retornará verdadero si la colección está vacía.
- Método `boolean contains (Object element)`: retornará verdadero si la colección tiene el elemento pasado como parámetro.
- Método `boolean add(E element)`: permitirá añadir elementos a la colección.
- Método `boolean remove (Object element)`: permitirá eliminar elementos de la colección.
- Método `Iterator<E> iterator()`: permitirá crear un iterador para recorrer los elementos de la colección. Esto se ve más adelante, no te preocupes.
- Método `Object[] toArray()`: permite pasar la colección a un array de objetos tipo `Object`.
- Método `containsAll(Collection<?> c)`: permite comprobar si una colección contiene los elementos existentes en otra colección, si es así, retorna verdadero.
- Método `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).
- Método `boolean removeAll(Collection<?> c)`: si los elementos de la colección pasada como parámetro están en nuestra colección, se eliminan, el resto se quedan.
- Método `boolean retainAll(Collection<?> c)`: si los elementos de la colección pasada como parámetro están en nuestra colección, se dejan, el resto se eliminan.
- Método `void clear()`: vaciar la colección.

Más adelante veremos como se usan estos métodos, será cuando veamos las implementaciones (clases genéricas que implementan alguna de las interfaces derivadas de la interfaz **Collection**).

## 2.- Conjuntos (I). Set



¿Con qué relacionarías los conjuntos? Seguro que con las matemáticas. Los conjuntos son un tipo de colección que **no admite duplicados**, derivados del concepto matemático de conjunto.

La interfaz `java.util.Set` define cómo deben ser los conjuntos, y extiende la interfaz `Collection`, aunque no añade ninguna operación nueva. Las implementaciones (clases genéricas que implementan la interfaz `Set`) más usadas son las siguientes:

- `java.util.HashSet`. Conjunto que almacena los objetos usando [tablas hash](#), lo cual acelera enormemente el acceso a los objetos almacenados. Inconvenientes: necesitan bastante memoria y **no almacenan los objetos de forma ordenada** (al contrario pueden aparecer completamente desordenados).
- `java.util.LinkedHashSet`. Conjunto que almacena objetos combinando tablas hash, para un acceso rápido a los datos, y [listas enlazadas](#) para conservar el orden. **El orden de almacenamiento es el de inserción**, por lo que se puede decir que es una estructura ordenada a medias. Inconvenientes: necesitan bastante memoria y es algo más lenta que `HashSet`.
- `java.util.TreeSet`. Conjunto que almacena los objetos usando unas estructuras conocidas como árboles rojo-negro. Son más lentas que los dos tipos anteriores, pero tienen una gran ventaja: **los datos almacenados se ordenan por valor**. Es decir, que aunque se inserten los elementos de forma desordenada, internamente se ordenan dependiendo del valor de cada uno.

No entraremos en profundidad que son las listas enlazadas y los árboles. Veamos un ejemplo de uso básico de la estructura `HashSet`.

Para [crear un conjunto](#), simplemente creamos el `HashSet` indicando el tipo de objeto que va a almacenar, dado que es una clase genérica que puede trabajar con cualquier tipo de dato debemos crearlo como sigue (no olvides hacer la importación de `java.util.HashSet` primero):

```
HashSet<Integer> conjunto = new HashSet<Integer>();  
HashSet<Integer> conjunto = new HashSet<>(); //OTRA FORMA DE CREACIÓN
```

Después podremos ir almacenando objetos dentro del conjunto usando el método `add` (definido por la interfaz `Set`). Los objetos que se pueden insertar serán siempre del tipo especificado al crear el conjunto:

```
Integer n = new Integer(10);  
if (!conjunto.add(n)) //equivale a (conjunto.add(n)==false)  
    System.out.println("Número ya en la lista.");
```

Si el elemento ya está en el conjunto, el método `add` retornará `false` indicando que no se pueden insertar duplicados. Si todo va bien, retornará `true`.

## 2.1.- Conjuntos (II).

---

Y ahora te preguntará, ¿cómo accedo a los elementos almacenados en un conjunto? Para obtener los elementos almacenados en un conjunto hay que usar iteradores, que permiten obtener los elementos del conjunto uno a uno de forma secuencial (no hay otra forma de acceder a los elementos de un conjunto, es su inconveniente). Los iteradores se ven en mayor profundidad más adelante, de momento, vamos a usar iteradores 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 elemento: conjunto) {  
    System.out.println("Elemento almacenado:"+elemento);  
}
```

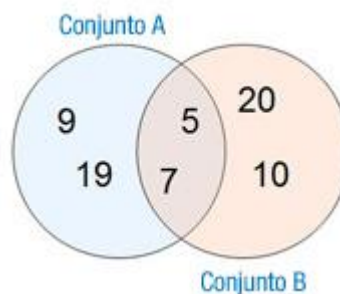
Como ves la estructura for-each es muy sencilla: la palabra **for** seguida de "(tipo variable:colección)" y el cuerpo del bucle; tipo es el tipo del objeto sobre el que se ha creado la colección, variable pues es la variable donde se almacenará cada elemento de la colección y colección pues la colección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

## 2.3.- Conjuntos (III).

---

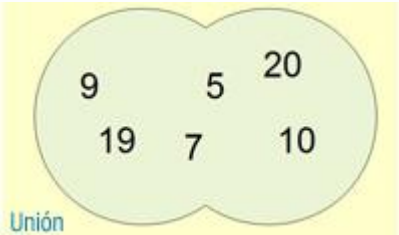


¿Cómo podría copiar los elementos de un conjunto de uno a otro? ¿Hay que usar un bucle **for** y recorrer toda la lista para ello? ¡Qué va! Para facilitar esta tarea, los conjuntos, y las colecciones en general, facilitan un montón de operaciones para poder combinar los datos de varias colecciones. Ya se vieron en un apartado anterior, aquí simplemente vamos a poner un ejemplo de su uso.

Partimos del siguiente ejemplo, en el que hay dos colecciones, cada una con 4 números enteros:



```
Set<Integer> cjoA= new HashSet<>();  
cjoA.add(9); cjoA.add(19); cjoA.add(5); cjoA.add(7);  
// Elementos del conjunto A: 9, 19, 5 y 7  
  
Set<Integer> cjoB= new HashSet<>();  
cjoB.add(10); cjoB.add(20); cjoB.add(5); cjoB.add(7);  
// Elementos del conjunto B: 10, 20, 5 y 7
```

En el ejemplo anterior, el literal de número se convierte automáticamente a la clase envoltorio **Integer** sin tener que hacer nada, lo cual es una ventaja. Veamos las formas de combinar ambas colecciones:

Tipos de combinaciones.		
Combinación.	Código.	Elementos finales del conjunto A.
Unión. Añadir todos los elementos del conjunto B en el conjunto A.	<code>cjoA.addAll(cjoB)</code>	<div><p>Todos los del conjunto A, añadiendo los del B, pero sin repetir los que ya están: 5, 7, 9, 10, 19 y 20.</p></div>
Diferencia. Eliminar los elementos del conjunto B que puedan estar en el conjunto A.	<code>cjoA.removeAll(cjoB)</code>	<div><p>Todos los elementos del conjunto A, que no estén en el conjunto B: 9, 19.</p></div>
Intersección. Retiene los elementos comunes a ambos conjuntos.	<code>cjoA.retainAll(cjoB)</code>	<div><p>Todos los elementos del conjunto A, que también están en el conjunto B: 5 y 7.</p></div>

Recuerda, estas operaciones son comunes a todas las colecciones.

## 3.- Listas (I). List

¿En qué se diferencia una lista de un conjunto? Las listas son elementos de programación un poco más avanzados que los conjuntos. Su ventaja es que amplían el conjunto de operaciones de las colecciones añadiendo operaciones extra, veamos algunas de ellas:

- 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 podía comprobar si un conjunto contenía 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 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).

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.

### 3.1.- Listas (II).

Y, ¿cómo se usan las listas? Pues para usar una lista haremos uso de sus implementaciones **LinkedList** y **ArrayList**. Veamos un ejemplo de su uso y después obtendrás respuesta a esta pregunta.

Supongo que intuirás como se usan, pero nunca viene mal un ejemplo sencillo, que nos aclare las ideas. El siguiente ejemplo muestra como usar un **LinkedList** pero valdría también para **ArrayList** (no olvides importar las clases `java.util.LinkedList` y `java.util.ArrayList` según sea necesario). En este ejemplo se usan los métodos de acceso posicional a la lista:

```
// Declaración y creación del LinkedList de enteros.
LinkedList<Integer> t = new LinkedList<Integer>();
LinkedList<Integer> t = new LinkedList<>(); //OTRA FORMA SIN REPETIR TIPO ALMACENADO

t.add(1); // Añade un elemento al final de la lista.
t.add(3); // Añade otro elemento al final de la lista.
t.add(1,2); // Añade en la posición 1 el elemento 2.
t.add(t.get(1)+t.get(2)); // Suma los valores contenidos en la posición 1 y 2, y lo
agrega al final.
t.remove(0); // Elimina el primer elementos de la lista.
for (Integer i: t)

    System.out.println("Elemento:" + i); // Muestra la lista.
```

En el ejemplo anterior, se realizan muchas operaciones, ¿cuál será el contenido de la lista al final? Pues será 2, 3 y 5. En el ejemplo cabe destacar el uso del bucle for-each, recuerda que se puede usar en cualquier colección.

Veamos otro ejemplo, esta vez con **ArrayList**, de cómo obtener la posición de un elemento en la lista:

```
ArrayList<Integer> al=new ArrayList<>(); // Declaración y creación del ArrayList de
enteros.
al.add(10); al.add(11); // Añadimos dos elementos a la lista.
al.set(al.indexOf(11), 12); // Sustituimos el 11 por el 12, primero lo buscamos y luego
lo reemplazamos.
```

En el ejemplo anterior, se emplea tanto el método **indexOf** para obtener la posición de un elemento, como el método **set** para reemplazar el valor en una posición, una combinación muy habitual. El ejemplo anterior generará un **ArrayList** que contendrá dos números, el 10 y el 12. Veamos ahora un ejemplo algo más difícil:

```
al.addAll(0, t.subList(1, t.size()));
```

Este ejemplo es especial porque usa sublistas. Se usa el método **size** para obtener el tamaño de la lista. Después el método **subList** para extraer una sublista de la lista (que incluía en origen los números 2, 3 y 5), desde la posición 1 hasta el final de la lista (lo cual dejaría fuera al primer elemento). Y por último, se usa el método **addAll** para añadir todos los elementos de la sublista al **ArrayList** anterior.

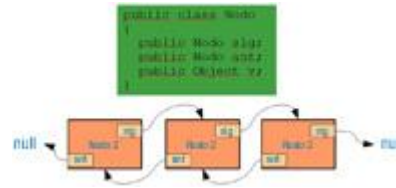
Debes saber que las operaciones aplicadas a una sublista repercuten sobre la lista original. Por ejemplo, si ejecutamos el método **clear** sobre una sublista, se borrarán todos los elementos de la sublista, pero también se borrarán dichos elementos de la lista original:

```
al.subList(0, 2).clear();
```

---

## 3.2.- Listas (III).

---



¿Y en qué se diferencia un **LinkedList** de un **ArrayList**? Los **LinkedList** utilizan listas doblemente enlazadas, que son listas enlazadas (como se vio en un apartado anterior), 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 redundaría 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.

¿Y esto que quiere decir? **Que 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).**

## 5.- Iteradores (I).

---

¿Qué son los iteradores realmente? Son un mecanismo que nos permite recorrer todos los elementos de una colección de forma sencilla, de forma secuencial, y de forma segura.

Los iteradores permiten recorrer las colecciones de dos formas:

- **bucles for-each** (existentes en Java a partir de la versión 1.5) **y**
- **un bucle normal creando un iterador.**

Como los bucles for-each ya los hemos visto antes (y ha quedado patente su simplicidad), nos vamos a centrar en el otro método, especialmente útil en versiones antiguas de Java. Ahora la pregunta es, ¿cómo se crea un iterador? Pues invocando el método "**iterator()**" de cualquier colección. Veamos un ejemplo (en el ejemplo **col** es una colección cualquiera):

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

Fijate que 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, igualmente nos permitiría 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 pasárselo por parámetro). Cuidado, si next no ha sido invocado todavía, saltará una incomoda excepción.

¿Cómo recorreríamos una colección con estos métodos? Pues de una forma muy sencilla, un simple bucle mientras (**while**) con la condición **hasNext()** 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? Efectivamente, solo los números impares.

---



## Recomendación

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, o dentro de un bucle `for-each`, los fallos que pueden producirse en tu programa son impredecibles. ¿Logras adivinar porqué se pueden producir dichos problemas?

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.

## 6.- Algoritmos (I).

La palabra algoritmo seguro que te suena, pero, ¿a qué se refiere en el contexto de las colecciones y de otras estructuras de datos? Las colecciones, los arrays e incluso las cadenas, tienen un conjunto de operaciones típicas asociadas que son habituales. Algunas de estas operaciones ya las hemos visto antes, pero otras no. Veamos para qué nos pueden servir estas operaciones:

- Ordenar listas y arrays.
- Desordenar listas y arrays.
- Búsqueda binaria en listas y arrays.
- Conversión de arrays a listas y de listas a array.

Estos algoritmos están en su mayoría recogidos como métodos estáticos de las **clases** `java.util.Collections` y `java.util.Arrays`, salvo los referentes a cadenas obviamente.

Los algoritmos de ordenación ordenan los elementos en orden natural, siempre que Java sepa como ordenarlos. Como se explico en el apartado de conjuntos, cuando se desea que la ordenación siga un orden diferente, o simplemente los elementos no son ordenables de forma natural, hay que facilitar un mecanismo para que se pueda producir la ordenación. Los tipos "ordenables" de forma natural son los enteros, las cadenas (orden alfabético) y las fechas, y por defecto su orden es ascendente.

La clase **Collections** y la clase **Arrays** facilitan el método **sort**, que permiten ordenar respectivamente listas y arrays. Los siguientes ejemplos ordenarían los números de forma ascendente (de menor a mayor):

### Ordenación natural en listas y arrays.

#### Ejemplo de ordenación de un array de números

```
Integer[] array={10,9,99,3,5};  
  
Arrays.sort(array);
```

#### Ejemplo de ordenación de una lista con números

```
ArrayList<Integer> lista=new ArrayList<>();  
lista.add(10); lista.add(9);lista.add(99);  
lista.add(3); lista.add(5);  
  
Collections.sort(lista);
```

## 6.2.- Algoritmos (II).

¿Qué más ofrece las **clases** `java.util.Collections` y `java.util.Arrays` de Java? Una vez vista la ordenación, quizás lo más complicado, veamos algunas operaciones adicionales. En los ejemplos, la variable "**array**" es un array y la variable "lista" es una lista de cualquier tipo de elemento:

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