

# Colecciones de datos en Java.



## Objetivos

En esta unidad estudiaremos las colecciones dinámicas, que se caracterizan porque no tienen un tamaño fijo, como los arrays. El número de elementos que tienen se pueden ir ampliando o disminuyendo a lo largo de la ejecución del programa según las necesidades y espacio disponible en memoria. Ocupan exactamente el tamaño que se necesita para los elementos que contienen:

- Si se elimina un elemento de la estructura, disminuye el tamaño necesario para el almacenamiento de la misma, liberando el espacio que ya no se necesita.
- Si se inserta un nuevo elemento, se busca espacio libre en memoria para él, y la estructura aumenta de tamaño.

Las diferentes colecciones dinámicas que existen se diferencian por la forma en la que se relacionan los diferentes datos que lo componen.

## 1.- Introducción

---

¿Qué consideras 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 hacer uso de estos algoritmos.

Las colecciones son en general elementos de programación que están disponibles en muchos lenguajes de programación.

## 2.- Uso de genéricos.

¿Sabes porqué se suele aprender el uso de los genéricos? Pues porque se necesita para usar las listas, aunque realmente los genéricos son una herramienta muy potente y que nos puede ahorrar tareas de programación repetitivas.

Las clases y los métodos genéricos son un recurso de programación disponible en muchos lenguajes de programación. Su objetivo es claro: facilitar la reutilización del software, creando métodos y clases que puedan trabajar con diferentes tipos de objetos, evitando incomodas y engorrosas conversiones de tipos. Su inicio se remonta a las plantillas (templates) de C++, un gran avance en el mundo de programación sin duda. En lenguajes de más alto nivel como Java o C# se ha transformado en lo que se denomina "genéricos".

Veamos un ejemplo sencillo de cómo transformar un método normal en genérico:

### Versión genérica y no genérica del método de compararTamano.

Versión no genérica	Versión genérica del método
<pre>public class util {     public static int compararTamano(Object[] a, Object[] b) {         return a.length-b.length;     } }</pre>	<pre>public class util {     public static &lt;T&gt; int compararTamano (T[] a, T[] b) {         return a.length-b.length;     } }</pre>

Los dos métodos anteriores tienen un claro objetivo: permitir comprobar si un array es mayor que otro. Retornarán 0 si ambos arrays son iguales, un número mayor de cero si el array **b** es mayor, y un número menor de cero si el array **a** es mayor, pero uno es genérico y el otro no.

La versión genérica del módulo incluye la expresión "<T>", justo antes del tipo returned por el método. "<T>" es la definición de una **variable o parámetro formal de tipo** de la clase o método genérico, al que podemos llamar simplemente **parámetro de tipo o parámetro genérico**. Este parámetro genérico (T) se puede usar a lo largo de todo el método o clase, dependiendo del ámbito de definición, y hará referencia a cualquier clase con la que nuestro algoritmo tenga que trabajar. Como veremos más adelante, puede haber más de un parámetro genérico.

Utilizar genéricos tiene claras ventajas. Para invocar un método genérico, sólo hay que realizar una **invocación de tipo genérico**, olvidándonos de las conversiones de tipo. Esto consiste en indicar qué clases o interfaces concretas se utilizarán en lugar de cada parámetro genérico ("<T>"), para después, pasándole los argumentos correspondientes, ejecutar el algoritmo. Cada clase o interfaz concreta, la podemos denominar **tipo o tipo base** y se da por sentado que **los argumentos pasados al método genérico serán también de dicho tipo base**.

Supongamos que el tipo base es **Integer**, pues para realizar la invocación del método genérico anterior basta con indicar el tipo, entre los símbolos de menor qué y mayor qué ("<Integer>"), justo antes del nombre del método.

### Invocaciones de las versiones genéricas y no genéricas de un método.

Invocación del método no genérico.	Invocación del método genérico.
Integer []a={0,1,2,3,4};	Integer []a={0,1,2,3,4};

Invocación del método no genérico.	Invocación del método genérico.
<pre>Integer []b={0,1,2,3,4,5}; util.compararTamano ((Object[])a, (Object[])b);</pre>	<pre>Integer []b={0,1,2,3,4,5}; util.&lt;Integer&gt;compararTamano (a, b);</pre>

## 2.1.- Clases y métodos genéricos.

¿Crees qué el código es más legible al utilizar genéricos o que se complica? La verdad es que al principio cuesta, pero después, el código se entiende mejor que si se empieza a insertar conversiones de tipo.

Las clases genéricas son equivalentes a los métodos genéricos pero a nivel de clase, permiten definir un parámetro de tipo o genérico que se podrá usar a lo largo de toda la clase, facilitando así crear clases genéricas que son capaces de trabajar con diferentes tipos de datos base. Para crear una clase genérica se especifican los parámetros de tipo al lado del nombre de la clase:

```
public class Util<T> {
    T t1;
    public void invertir(T[] array) {
        for (int i = 0; i < array.length / 2; i++) {
            t1 = array[i];
            array[i] = array[array.length - i - 1];
            array[array.length - i - 1] = t1;
        }
    }
}
```

En el ejemplo anterior, la clase **Util** contiene el método **invertir** cuya función es invertir el orden de los elementos de cualquier array, sea del tipo que sea. Para usar esa clase genérica hay que crear un objeto o instancia de dicha clase especificando el tipo base entre los símbolos menor que ("<") y mayor que (">"), justo detrás del nombre de la clase. Veamos un ejemplo:

```
Integer[] numeros={0,1,2,3,4,5,6,7,8,9};
Util<Integer> u= new Util<Integer>();
u.invertir(numeros);
for (int i=0;i<numeros.length;i++)
    System.out.println(numeros[i]);
```

Como puedes observar, el uso de genéricos es sencillo, tanto a nivel de clase como a nivel de método. Simplemente, a la hora de crear una instancia de una clase genérica, hay que especificar el tipo, tanto en la definición (**Util <integer> u**) como en la creación (**new Util<Integer>()**).

Los genéricos los vamos a usar ampliamente a partir de ahora, aplicados a un montón de clases genéricas que tiene Java y que son de gran utilidad, por lo que es conveniente que aprendas bien a usar una clase genérica.

Los parámetros de tipo de las clases genéricas solo pueden ser clases, no pueden ser jamás tipos de datos primitivos como **int**, **short**, **double**, etc. En su lugar, debemos usar sus clases envoltorio **Integer**, **Short**, **Double**, etc.



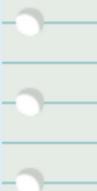
**Debes conocer**

Todavía hay un montón de cosas más sobre los métodos y las clases genéricas que deberías saber. En la siguiente animación se muestran algunos usos interesantes de los genéricos:



## Autoevaluación

Dada la siguiente clase, donde el código del método prueba carece de importancia, ¿podrías decir cuál de las siguientes invocaciones es la correcta?



```
public class Util {  
    public static <T> int prueba (T t) { ... }  
};
```

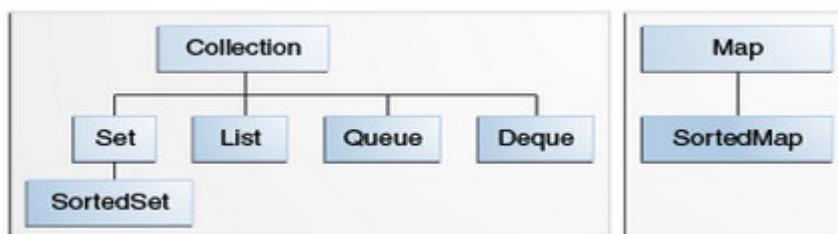
- Util.<int>prueba(4);
- Util.<Integer>prueba(new Integer(4));
- Util u=new Util(); u.<int>prueba(4);

### 3.- Introducción a las colecciones.

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`).

## 4.- Conjuntos.

¿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 almacenado. 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 mas 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.

Poco a poco, iremos viendo que son las listas enlazadas y los árboles (no profundizaremos en los árboles rojo-negro, pero si veremos las estructuras tipo árbol en general). Veamos un ejemplo de uso básico de la estructura `HashSet` y después, profundizaremos en los `LinkedHashSet` y los `TreeSet`.

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>();
```

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))
    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`.



### Autoevaluación

¿Cuál de las siguientes estructuras ordena automáticamente los elementos según su valor?

- `HashSet`.

• **LinkedHashSet.**

• **TreeSet.**

## 4.1.- Acceso a elementos.

Y ahora te preguntarás, ¿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 i: conjunto) {  
    System.out.println("Elemento almacenado:" + i);  
}
```

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.



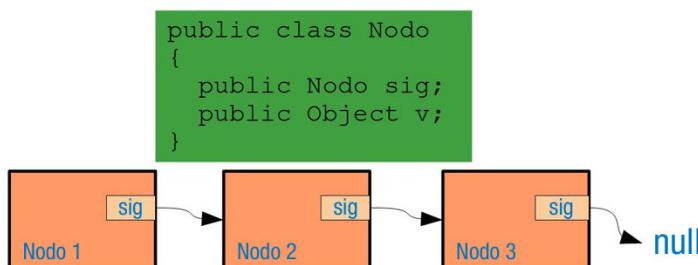
### Ejercicio resuelto

Realiza un pequeño programita que pregunte al usuario 5 números diferentes (almacenándolos en un **HashSet**), y que después calcule la suma de los mismos (usando un bucle for-each).

## 4.2.- Características.

¿En qué se diferencian las estructuras `LinkedHashSet` y `TreeSet` de la estructura `HashSet`? Ya se comentó antes, y es básicamente en su funcionamiento interno.

La estructura `LinkedHashSet` es una estructura que internamente funciona como una lista enlazada, aunque usa también tablas hash para poder acceder rápidamente a los elementos. Una lista enlazada es una estructura similar a la representada en la imagen de abajo, la cual está compuesta por nodos (elementos que forman la lista) que van enlazándose entre sí. Un nodo contiene dos cosas: el dato u objeto almacenado en la lista y el siguiente nodo de la lista. Si no hay siguiente nodo, se indica poniendo nulo (`null`) en la variable que contiene el siguiente nodo.

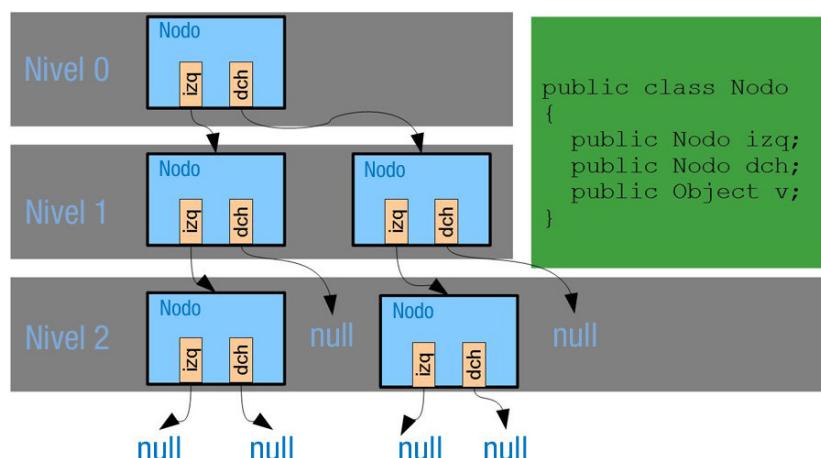


Las listas enlazadas tienen un montón de operaciones asociadas en las que no vamos a profundizar: eliminación de un nodo de la lista, inserción de un nodo al final, al principio o entre dos nodos, etc. Gracias a las colecciones podremos utilizar listas enlazadas sin tener que complicarnos en detalles de programación.

La estructura `TreeSet`, en cambio, utiliza internamente árboles. Los árboles son como las listas pero mucho más complejos. En vez de tener un único elemento siguiente, pueden tener dos o más elementos siguientes, formando estructuras organizadas y jerárquicas.

Los nodos se diferencian en dos tipos: nodos padre y nodos hijo; un nodo padre puede tener varios nodos hijo asociados (depende del tipo de árbol), dando lugar a una estructura que parece un árbol invertido (de ahí su nombre).

En la siguiente figura se puede apreciar un árbol donde cada nodo puede tener dos hijos, denominados izquierdo (`izq`) y derecho (`dch`). Puesto que un nodo hijo puede también ser padre a su vez, los árboles se suelen visualizar para su estudio por niveles para entenderlos mejor, donde cada nivel contiene hijos de los nodos del nivel anterior, excepto el primer nivel (que no tiene padre).



Los árboles son estructuras complejas de manejar y que permiten operaciones muy sofisticadas. Los árboles usados en los `TreeSet`, los árboles rojo-negro, son árboles auto-ordenados, es decir, que al insertar un elemento, este queda ordenado por su valor de forma que al recorrer el árbol, pasando por todos los nodos, los elementos salen ordenados. El ejemplo mostrado en la imagen es simplemente un árbol binario, el más simple de todos.

Nuevamente, no se va a profundizar en las operaciones que se pueden realizar en un árbol a nivel interno

(inserción de nodos, eliminación de nodos, búsqueda de un valor, etc.). Nos aprovecharemos de las colecciones para hacer uso de su potencial.

En la siguiente tabla tienes un uso comparado de **TreeSet** y **LinkedHashSet**. Su creación es similar a como se hace con **HashSet**, simplemente sustituyendo el nombre de la clase **HashSet** por una de las otras. Ni **TreeSet**, ni **LinkedHashSet** admiten duplicados, y se usan los mismos métodos ya vistos antes, los existentes en la interfaz **Set** (que es la interfaz que implementan).

### Ejemplos de utilización de los conjuntos **TreeSet** y **LinkedHashSet**

	Conjunto <b>TreeSet</b> .	Conjunto <b>LinkedHashSet</b> .
Ejemplo de uso	<pre>TreeSet &lt;Integer&gt; t; t=new TreeSet&lt;Integer&gt;(); t.add(new Integer(4)); t.add(new Integer(3)); t.add(new Integer(1)); t.add(new Integer(99)); for (Integer i:t) System.out.println(i);</pre>	<pre>LinkedHashSet &lt;Integer&gt; t; t=new LinkedHashSet&lt;Integer&gt;(); t.add(new Integer(4)); t.add(new Integer(3)); t.add(new Integer(1)); t.add(new Integer(99)); for (Integer i:t) System.out.println(i);</pre>
Resultado mostrado por pantalla	1 3 4 99 (el resultado sale ordenado por valor)	4 3 1 99 (los valores salen ordenados al azar)



### Autoevaluación

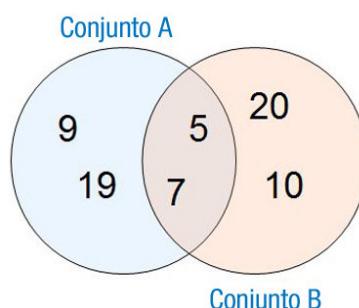
Un árbol cuyos nodos solo pueden tener un único nodo hijo, en realidad es una lista.  
¿Verdadero o falso?

- Verdadero.
- Falso.

## 4.3.- Combinaciones.

¿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 poner un ejemplo de su uso.

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

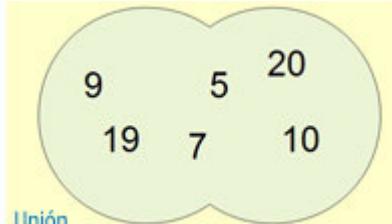
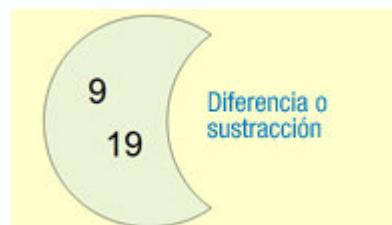


```
TreeSet<Integer> A= new TreeSet<Integer>();
A.add(9); A.add(19); A.add(5); A.add(7); // Elementos del conjunto A: 9, 19, 5 y 7
LinkedHashSet<Integer> B= new LinkedHashSet<Integer>();
B.add(10); B.add(20); B.add(5); B.add(7); // Elementos del conjunto B: 10, 20, 5 y 7
```

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.
<b>Unión.</b> <b>Añadir todos los elementos del conjunto B en el conjunto A.</b>	<code>A.addAll(B)</code>	Todos los del conjunto A, añadiendo los del B, pero sin repetir los que ya están: 5, 7, 9, 10, 19 y 20.  <b>Unión</b>
<b>Diferencia.</b> <b>Eliminar los elementos del conjunto B que puedan estar en el conjunto A.</b>	<code>A.removeAll(B)</code>	Todos los elementos del conjunto A, que no estén en el conjunto B: 9, 19.  <b>Diferencia o sustracción</b>

Combinación.	Código.	Elementos finales del conjunto A.
<p><b>Intersección.</b></p> <p>Retiene los elementos comunes a ambos conjuntos.</p>	A.retainAll(B)	<p>Todos los elementos del conjunto A, que también están en el conjunto B: 5 y 7.</p> 

Recuerda, estas operaciones son comunes a todas las colecciones.



## Autoevaluación

Tienes un **HashSet** llamado **vocales** que contiene los elementos "a", "e", "i", "o", "u", y otro, llamado **vocales\_fuertes** con los elementos "a", "e" y "o". ¿De qué forma podríamos sacar una lista con las denominadas vocales débiles (que son aquellas que no son fuertes)?

- `vocales.retainAll (vocales_fuertes);`
- `vocales.removeAll(vocales_fuertes);`
- No es posible hacer esto con **HashSet**, solo se puede hacer con **TreeSet** o **LinkedHashSet**.

## 4.4.- Ordenaciones.

Por defecto, los `TreeSet` ordenan sus elementos de forma ascendente, pero, ¿se podría cambiar el orden de ordenación? Los `TreeSet` tienen un conjunto de operaciones adicionales, además de las que incluye por el hecho de ser un conjunto, que permite entre otras cosas, cambiar la forma de ordenar los elementos. Esto es especialmente útil cuando el tipo de objeto que se almacena no es un simple número, sino algo más complejo (un artículo por ejemplo). `TreeSet` es capaz de ordenar tipos básicos (números, cadenas y fechas) pero otro tipo de objetos no puede ordenarlos con tanta facilidad.

Para indicar a un `TreeSet` cómo tiene que ordenar los elementos, debemos decirle cuándo un elemento va antes o después que otro, y cuándo son iguales. Para ello, utilizamos la interfaz genérica `java.util.Comparator`, usada en general en algoritmos de ordenación, como veremos más adelante. Se trata de crear una clase que implemente dicha interfaz, así de fácil. Dicha interfaz requiere de un único método que debe calcular si un objeto pasado por parámetro es mayor, menor o igual que otro del mismo tipo. Veamos un ejemplo general de cómo implementar un comparador para una hipotética clase "`Objeto`":

```
class ComparadorDeObjetos implements Comparator<Objeto> {
    public int compare(Objeto o1, Objeto o2) { ... }
}
```

La interfaz `Comparator` obliga a implementar un único método, es el método `compare`, el cual tiene dos parámetros: los dos elementos a comparar. Las reglas son sencillas, a la hora de personalizar dicho método:

- ✓ Si el primer objeto (`o1`) es menor que el segundo (`o2`), debe retornar un número entero negativo.
- ✓ Si el primer objeto (`o1`) es mayor que el segundo (`o2`), debe retornar un número entero positivo.
- ✓ Si ambos son iguales, debe retornar 0.

A veces, cuando el orden que deben tener los elementos es diferente al orden real (por ejemplo cuando ordenamos los números en orden inverso), la definición de antes puede ser un poco liosa, así que es recomendable en tales casos pensar de la siguiente forma:

- ✓ Si el primer objeto (`o1`) debe ir antes que el segundo objeto (`o2`), retornar entero negativo.
- ✓ Si el primer objeto (`o1`) debe ir después que el segundo objeto (`o2`), retornar entero positivo.
- ✓ Si ambos son iguales, debe retornar 0.

Una vez creado el comparador simplemente tenemos que pasarlo como parámetro en el momento de la creación al `TreeSet`, y los datos internamente mantendrán dicha ordenación:

```
TreeSet<Objeto> ts=new TreeSet<Objeto>(new ComparadorDeObjetos());
```



### Ejercicio resuelto

¿Fácil no? Pongámoslo en práctica. Imagínate que `Objeto` es una clase como la siguiente:

```
class Objeto {
```

```
public int a;  
public int b;  
}
```

Imagina que ahora, al añadirlos en un `TreeSet`, estos se tienen que ordenar de forma que la suma de sus atributos (`a` y `b`) sea descendente, ¿como sería el comparador?

## 5.- Listas.

¿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.



### Autoevaluación

**Si M es una lista de números enteros, ¿sería correcto poner "M.add(M.size(),3);"**

- Sí.
- No.

## 5.1.- Uso.

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 cómo 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:

```
LinkedList<Integer> t=new LinkedList<Integer>(); // Declaración y creación del LinkedList de enteros.
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<Integer>(); // 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();
```

Lo mismo ocurre al añadir un elemento, se añade en la sublistas y en la lista original.



## Debes conocer

Las listas enlazadas son un elemento muy recurrido y su funcionamiento interno es complejo. Te recomendamos el siguiente artículo de la wikipedia para profundizar un poco más en las listas enlazadas y los diferentes tipos que hay.

[Listas enlazadas.](#)



## Autoevaluación

Completa con el número que falta.

Dado el siguiente código:

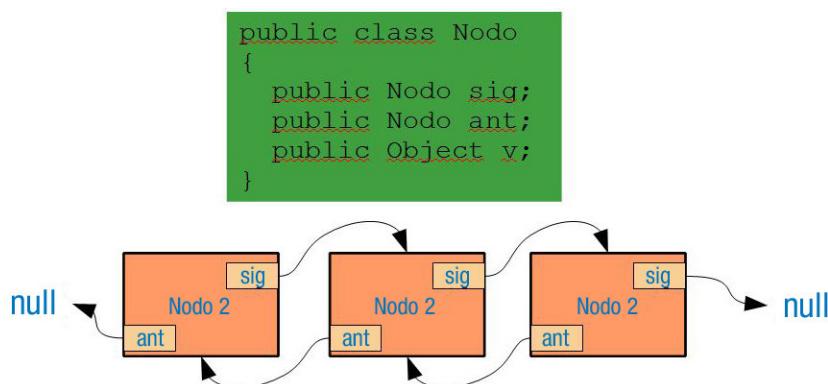
```
- LinkedList<Integer> t=new LinkedList<Integer>();  
- t.add(t.size()+1);  
- t.add(t.size());  
- Integer suma = t.get(0) + t.get(1);
```

El valor de la variable suma después de ejecutarlo es  .

## 5.2.- Características.

¿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 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.

¿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).**

**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 pero que aportan métodos para trabajar de forma diferente. ¿Tú sabes lo que es hacer cola para que te atiendan en una ventanilla? Pues igual. Se 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 listas. Una pila es igual que una montaña de hojas en blanco, para añadir hojas nuevas se ponen encima del resto, y para retirar una se coge la primera que hay, encima de todas. 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).

Las pilas se usan menos y haremos menos hincapié en ellas. Simplemente ten en mente que, tanto las colas como las pilas, son una lista enlazada sobre la que se hacen operaciones especiales.



## Autoevaluación

Dada la siguiente lista, usada como si fuera una cola de prioridad, ¿cuál es la letra que se mostraría por la pantalla tras su ejecución?

```
LinkedList<String> tt=new LinkedList<String>();  
tt.offer("A");  
tt.offer("B");  
tt.offer("C");  
System.out.println(tt.poll());
```

- A.
- C.
- D.

## 5.3.- Detalles.

---

A la hora de usar las listas, hay que tener en cuenta un par de detalles, ¿sabes cuáles? Es sencillo, pero importante.

No es lo mismo usar las colecciones (listas y conjuntos) con objetos inmutables (**Strings**, **Integer**, etc.) que con objetos mutables. Los objetos inmutables no pueden ser modificados después de su creación, por lo que cuando se incorporan a la lista, a través de los métodos **add**, se pasan por copia (es decir, se realiza una copia de los mismos). En cambio los objetos mutables (como las clases que tú puedes crear), no se copian, y eso puede producir efectos no deseados.

Imaginate la siguiente clase, que contiene un número:

```
class Test {
    public Integer num;
    Test (int num) {
        this.num=new Integer(num);
    }
}
```

La clase de antes es mutable, por lo que no se pasa por copia a la lista. Ahora imagina el siguiente código en el que se crea una lista que usa este tipo de objeto, y en el que se insertan dos objetos:

```
Test p1=new Test(11); // Se crea un objeto Test donde el entero que contiene vale 11.
Test p2=new Test(12); // Se crea otro objeto Test donde el entero que contiene vale 12.
LinkedList<Test> lista=new LinkedList<Test>(); // Creamos una lista enlazada para objetos tipo Test.
lista.add(p1); // Añadimos el primero objeto test.
lista.add(p2); // Añadimos el segundo objeto test.
for (Test p:lista)
    System.out.println(p.num); // Mostramos la lista de objetos.
```

¿Qué mostraría por pantalla el código anterior? Simplemente mostraría los números 11 y 12. Ahora bien, ¿qué pasa si modificamos el valor de uno de los números de los objetos test? ¿Qué se mostrará al ejecutar el siguiente código?

```
p1.num=44;
for (Test p:lista)
    System.out.println(p.num);
```

El resultado de ejecutar el código anterior es que se muestran los números 44 y 12. El número ha sido modificado y no hemos tenido que volver a insertar el elemento en la lista para que en la lista se cambie también. Esto es porque en la lista no se almacena una copia del objeto **Test**, sino un apuntador a dicho objeto (solo hay una copia del objeto a la que se hace referencia desde distintos lugares).



## Autoevaluación

**Los elementos de un ArrayList de objetos Short se copian al insertarse al ser objetos mutables. ¿Verdadero o falso?**

- Verdadero.
- Falso.

## 6.- Conjuntos de pares clave/valor.

¿Cómo almacenarías 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`. ¿Te suenan? Claro que sí. Cada una de ellas, respectivamente, tiene características similares a `HashSet`, `TreeSet` y `LinkedHashSet`, tanto en funcionamiento interno como en rendimiento.

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



### Autoevaluación

Completa el siguiente código para que al final se muestre el número 40 por pantalla:

```
HashMap<String, [REDACTED]> datos=new [REDACTED]<String, String>();  
datos.[REDACTED](A,44);  
System.out.println(Integer.[REDACTED](datos.[REDACTED]([REDACTED])- [REDACTED]));
```

Enviar

## 7.- Iteradores.

¿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 mapas, como no derivan de la interfaz Collection realmente, no tienen iteradores, pero como veremos, existe un truco interesante.

Los iteradores permiten recorrer las colecciones de dos formas: **bucles for-each** (existentes en Java a partir de la versión 1.5) **y a través de 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 t es una colección cualquiera):



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

Fíjate 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 pasarselo por parámetro). Cuidado, si next no ha sido invocado todavía, saltará una incómoda 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.



## Reflexiona

Las listas permiten acceso posicional a través de los métodos `get` y `set`, y acceso secuencial a través de iteradores, ¿cuál es para tí la forma más cómoda 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())`"?

## 7.1.- Iteradores (II).

¿Qué inconvenientes tiene usar los iteradores sin especificar el tipo de objeto?

En el siguiente ejemplo, se genera una lista con los números del 0 al 10. De la lista, se eliminan aquellos que son pares y solo se dejan los impares. En el ejemplo de la izquierda se especifica el tipo de objeto del iterador, en el ejemplo de la derecha no, observa el uso de la conversión de tipos en la línea 5.

### Comparación de usos de los iteradores, con o sin conversión

Ejemplo indicando el tipo de objeto de iterador	Ejemplo no indicando el tipo de objeto de iterador
<pre>ArrayList &lt;Integer&gt; lista=new ArrayList&lt;Integer&gt;(); for (int i=0;i&lt;10;i++) lista.add(i); Iterator&lt;Integer&gt; it=lista.iterator(); while (it.hasNext()) {     Integer t=it.next();     if (t%2==0) it.remove(); }</pre>	<pre>ArrayList &lt;Integer&gt; lista=new ArrayList(); for (int i=0;i&lt;10;i++) lista.add(i); Iterator it=lista.iterator(); while (it.hasNext()) {     Integer t=(Integer)it.next();     if (t%2==0) it.remove(); }</pre>

Un iterador es seguro porque está pensado para no sobrepasar los límites de la colección, ocultando operaciones más complicadas que pueden repercutir en errores de software. Pero realmente se convierte en inseguro 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 incomoda excepción. Usar genéricos aporta grandes ventajas, pero usándolos 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 llaves existentes en el mapa. Veamos como sería para el segundo caso, el más sencillo:

<pre>HashMap&lt;Integer, Integer&gt; mapa=new HashMap&lt;Integer, Integer&gt;(); for (int i=0;i&lt;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. }</pre>
---

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 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 interíormente 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.



## Autoevaluación

¿Cuándo debemos invocar el método `remove()` de los iteradores?

- En cualquier momento.
- Despues de invocar el método `next()`.
- Despues de invocar el método `hasNext()`.
- No es conveniente usar este método para eliminar elementos, es mejor usar el de la colección.

## 8.- Algoritmos.

Las colecciones, 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.
- ✓ Desordenar listas.
- ✓ Búsqueda binaria en listas.
- ✓ 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**.

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	Ejemplo de ordenación de una lista c
<pre>Integer[] array={10,9,99,3,5}; Arrays.sort(array);</pre>	<pre>ArrayList&lt;Integer&gt; lista=new ArrayList&lt;Integer&gt;(); lista.add(10); lista.add(9); lista.add(99); lista.add(3); lista.add(5); Collections.sort(lista);</pre>

## 8.1.- Ordenaciones.

En Java hay dos mecanismos para cambiar la forma en la que los elementos se ordenan. Imagina que tienes los artículos de un pedido almacenados en una lista llamada "articulos", y que cada artículo se almacena en la siguiente clase (fíjate que el código de artículo es una cadena y no un número):

```
class Articulo {
    public String codArticulo;      // Código de artículo
    public String descripcion;     // Descripción del artículo.
    public int cantidad;           // Cantidad a proveer del artículo.
}
```

La primera forma de ordenar consiste en crear una clase que implemente la interfaz `java.util.Comparator`, y en ende, el método `compare` definido en dicha interfaz. Esto se explicó en el apartado de conjuntos, al explicar el `TreeSet`, así que no vamos a profundizar en ello. No obstante, el comparador para ese caso podría ser así:

```
class comparadorArticulos implements Comparator<Articulo> {
    @Override
    public int compare( Articulo o1, Articulo o2 ) {
        return o1.codArticulo.compareTo(o2.codArticulo);
    }
}
```

Una vez creada esta clase, ordenar los elementos es muy sencillo, simplemente se pasa como segundo parámetro del método `sort` una instancia del comparador creado:

```
Collections.sort(articulos, new comparadorArticulos());
```

La segunda forma es quizás más sencilla cuando se trata de objetos cuya ordenación no existe de forma natural, pero requiere modificar la clase `Articulo`. Consiste en hacer que los objetos que se meten en la lista o array implementen la interfaz `java.util.Comparable`. **Todos los objetos que implementan la interfaz Comparable son "ordenables" y se puede invocar el método sort sin indicar un comparador para ordenarlos.** La interfaz `comparable` solo requiere implementar el método `compareTo`:

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

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

Del ejemplo anterior se pueden denotar dos cosas importantes: que la interfaz Comparable es genérica y que para que funcione sin problemas es conveniente indicar el tipo base sobre el que se permite la comparación (en este caso, el objeto Artículo debe compararse consigo mismo), y que el método compareTo solo admite un parámetro, dado que comparará el objeto con el que se pasa por parámetro.

El funcionamiento del método compareTo es el mismo que el método compare de la interfaz Comparator: si la clase que se pasa por parámetro es igual al objeto, se tendría que retornar 0; si es menor o anterior, se debería retornar un número menor que cero; si es mayor o posterior, se debería retornar un número mayor que 0.

Ordenar ahora la lista de artículos es sencillo, fíjate que fácil: "Collections.sort(articulos);"



## Autoevaluación

Si tienes que ordenar los elementos de una lista de tres formas diferentes, ¿cuál de los métodos anteriores es más conveniente?

- Usar comparadores, a través de la interfaz java.util.Comparator.
- Implementar la interfaz comparable en el objeto almacenado en la lista.

## 8.2.- Otras operaciones.

¿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 arrays.

Operación	Descripción	Ejemplos
<b>Desordenar una lista.</b>	Desordena una lista, este método no está disponible para arrays.	<code>Collections.shuffle (lista);</code>
<b>Rellenar una lista o array.</b>	Rellena una lista o array copiando el mismo valor en todos los elementos del array o lista. Útil para reiniciar una lista o array.	<code>Collections.fill (lista,elemento);</code> <code>Arrays.fill (array,elemento);</code>
<b>Búsqueda binaria.</b>	Permite realizar búsquedas rápidas en una lista o array ordenados. Es necesario que la lista o array estén ordenados, si no lo están, la búsqueda no tendrá éxito.	<code>Collections.binarySearch(lista,elemento);</code> <code>Arrays.binarySearch(array, elemento);</code>
<b>Convertir un array a lista.</b>	Permite rápidamente convertir un array a una lista de elementos, extremadamente útil. No se especifica el tipo de lista retornada (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=Arrays.asList(array);</code>  Si el tipo de dato almacenado en el array es conocido (Integer por ejemplo), es conveniente especificar el tipo de objeto de la lista: <code>List&lt;Integer&gt;lista = Arrays.asList(array);</code>
<b>Convertir una lista a array.</b>	Permite convertir una lista a array. 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 array del tipo almacenado en la lista, y del tamaño de la lista: <code>Integer[] array=new Integer[lista.size()];</code> <code>lista.toArray(array)</code>
<b>Dar la vuelta.</b>	Da la vuelta a una lista, poniéndola en orden inverso al que tiene.	<code>Collections.reverse(lista);</code>