

Estructuras de datos internas (memoria)

Caso práctico



Ana ha recibido un pequeño encargo de parte de su tutora, **María**. Se trata de que realice un pequeño programita, muy sencillo pero fundamental.

—Hola **Ana**, hoy tengo una tarea especial para ti.

—¿Sí? Estoy deseando, últimamente no hay nada que se me resista, llevo dos semanas en racha —comenta **Ana**.

—Bueno, quizás esto se te resista un poco más, es fácil, pero tiene cierta complicación. En la aplicación que estás realizando para el cliente, necesitamos poder manejar datos de diferente tipo de una forma ágil y dinámica, manipular algo más que la información de un objeto o instancia concreta de datos... Necesitamos que la aplicación pueda guardar y manejar con facilidad conjuntos más grandes de datos complejos, como los pedidos que recibe, y Java aporta para eso algunas herramientas muy potentes. ¿Has oído hablar de Colecciones, Listas, Conjuntos, Mapas,...?

—Sí que me suena, pero no lo he usado antes... ¿Facilita el trabajo o lo complica? —pregunta **Ana**.

—¡¡Lo facilita, por supuesto!! No te preocupes, a partir de ahora vas a practicar con todo ello para esta aplicación y vas a entender hasta qué punto simplifica las cosas. Te voy poniendo al día de lo que debes hacer.

Cuando el volumen de datos a manejar por una aplicación es elevado, no basta con utilizar variables. Manejar los datos de un único pedido en una aplicación puede ser relativamente sencillo, pues un pedido está compuesto por una serie de datos y eso simplemente se traduce en varias variables. Pero, ¿qué ocurre cuando en una aplicación tenemos que gestionar varios pedidos a la vez?



[janeb13](#) (Licencia Pixabay)

Lo mismo ocurre en otros casos. Para poder realizar ciertas aplicaciones se necesita poder manejar datos que van más allá de meros **datos simples** (números y letras). A veces, los datos que tiene que manejar la aplicación son **datos compuestos**, es decir, datos que están compuestos a su vez de varios datos más simples. Por ejemplo, un pedido está compuesto por varios datos, que podrían ser el cliente que hace el pedido, la dirección de entrega, la fecha requerida de entrega y los artículos del pedido.

Los datos compuestos son un tipo de estructura de datos, y en realidad ya los has manejado. **Las clases son un ejemplo de estructuras de datos que permiten almacenar datos compuestos**, y el objeto en sí, la instancia de una clase, sería el dato compuesto. Pero, a veces, los datos tienen estructuras aún más complejas, y son necesarias soluciones adicionales.

Aquí podrás aprender esas soluciones adicionales. Esas soluciones consisten básicamente en la capacidad de poder manejar varios datos del mismo o diferente tipo de forma dinámica y flexible.

1.- Clases y métodos genéricos (I)

Caso práctico

María se acerca a la mesa de **Ana**, quiere saber cómo lo lleva:

—¿Qué tal? ¿Cómo vas con la tarea? — pregunta **María**.

—Bien, creo. Mi programita ya sabe procesar el archivo de pedido y he creado un par de clases para almacenar los datos de forma estructurada, pero no sé cómo almacenar los artículos del pedido, porque son varios —comenta **Ana**.

—Pero, ¿cuál es el problema? Eso es algo sencillo.

—Pues que tengo que crear un array para guardar en él los artículos del pedido, y no sé cómo averiguar el número de artículos antes de empezar a procesarlos. Es necesario saber el número de artículos para crear el array del tamaño adecuado.

—Pues en vez de utilizar un array, podrías utilizar **una lista**. Java proporciona maneras muy flexibles de solucionar esto, usando "genéricos" como por ejemplo ArrayList.

—Suená bien. Tendré que informarme y probarlo.



[Picography \(Licencia Pixabay\)](#)

¿Sabes por qué 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. Su objetivo es claro: facilitar la reutilización del software, creando métodos y clases que puedan trabajar con diferentes tipos de objetos, evitando incómodas 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:



Versiones 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 <T> int compararTamano (T[] a, T[] b) { return a.length-b.length; } }</pre>

Los dos métodos anteriores tienen el mismo objetivo: permitir comprobar si un array es más grande que otro (su longitud es mayor). Retornarán 0 si ambos arrays tienen la misma longitud, un número mayor de cero si el array a es más grande (su longitud es mayor), y un número menor de cero si el array b es más grande. La diferencia está en que uno de los métodos es genérico y el otro no. La versión genérica del método incluye la expresión "<T>", justo antes del tipo retornado 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 que y mayor que ("<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.
<pre>Integer[] a = {0,1,2,3,4}; Integer[] b = {0,1,2,3,4,5}; Util.compararTamano ((Object[])a, (Object[])b);</pre>	<pre>Integer[] a = {0,1,2,3,4}; Integer[] b = {0,1,2,3,4,5}; Util.<Integer>compararTamano (a, b);</pre>

1.1.- Clases y métodos genéricos (II)

¿Crees que 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:



[flag](#) (Licencia Pixabay)

```
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 - 1 - i];  
            array[array.length - 1 - i] = 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 ello, va intercambiando entre sí el primer elemento con el último (cuando i=0), el segundo con el penúltimo (cuando i=1), etc., hasta llegar a la mitad del array, momento en que se habrán intercambiado todos y el array estará invertido.

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>()). Así, el objeto u es una instancia de la clase genérica Util particularizada para objetos de tipo Integer. Por eso podemos invocar al método u.invertir(numeros), de forma que el parámetro formal T[] array (un array de objetos de tipo T) se sustituye por el parámetro actual numeros, que es un array de objetos de tipo Integer.

Ahora bien, a partir de Java 7 es posible utilizar el **operador diamante** ("<>") para simplificar la instanciación o creación de nuevos objetos a partir de clases genéricas, de manera que la instanciación anterior podría quedar como new Util<>(), sin necesidad de especificar el tipo "concreto" Integer en la llamada al constructor. El ejemplo anterior completo quedaría entonces como sigue:

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

Por tanto, a partir de ahora, **siempre que nos sea posible utilizaremos el operador diamante** ("<>") para mejorar la legibilidad del código, aunque la otra opción es siempre posible (y necesaria si por alguna razón se fuera a trabajar con versiones anteriores a Java 7).

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.

Debes conocer

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

 [Profundizando en las clases y métodos genéricos](#) (pdf - 82,72 KB).

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.

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);

2.- Introducción a las colecciones

Caso práctico



[Pexels \(Licencia Pixabay\)](#)

A **Ana** las listas siempre se le han atragantado, por eso procura evitar usarlas, y eso la lleva a perder una gran cantidad de tiempo "inventando la rueda" para conseguir lo que podría obtener de forma casi automática si aprovechara la potencia que éstas proporcionan, sobre todo en el lenguaje Java.

Después de darle muchas vueltas, ha pensado que no le queda más remedio y que tendrá que usarlas para almacenar los artículos del pedido. Además, se ha convencido que es la mejor forma de gestionar un grupo de objetos, aunque sean del mismo tipo.

No sabe si lo más adecuado es usar una lista u otro tipo de colección, así que ha decidido revisar todos los tipos de colecciones disponibles en Java, para ver cuál se adecua mejor a sus necesidades.

Así que se toma un descanso para mentalizarse, y atacar la cuestión con ánimos renovados.

¿Qué es para ti una colección?

Seguramente al pensar en el término se te viene a la cabeza una colección de libros, o de mariposas, 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. 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. En algunos de ellos 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 unidad.

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()`: devuelve el número de elementos de la colección.
- ✔ Método `boolean isEmpty()`: devuelve `true` si la colección está vacía.
- ✔ Método `boolean contains (Object element)`: retornará `true` 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 cómo se usan estos métodos, será cuando veamos las implementaciones (clases genéricas que implementan alguna de las interfaces derivadas del interface `Collection`).

3.- Conjuntos (I). ¿Qué son y cómo se declaran? HashSet

Caso práctico

Ana se levanta para ir a sacar un café de la máquina, y en el pasillo se encuentra con **Juan**, con el que entabla una conversación bastante amena. Una cosa lleva a otra y al final, **Ana** saca el tema que más le preocupa:

—¿Cuántos tipos de colecciones hay? ¿Tú lo sabes?
—pregunta **Ana**.

—¿Yo? ¡Qué va! Normalmente consulto la documentación cuando las voy a usar, como todo el mundo. Lo que sí creo recordar es que había cuatro tipos básicos: los **conjuntos**, las **listas**, las **colas** y alguno más que no recuerdo. ¡Ah, sí!, los **mapas**, aunque creo que no se consideraban un tipo de colección. ¿Por qué lo preguntas?

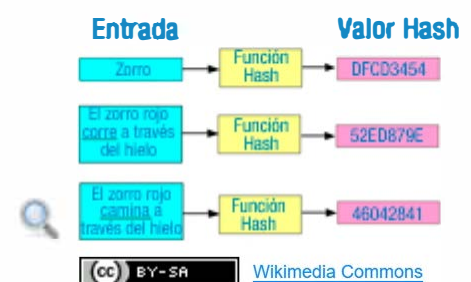
—Pues porque tengo que usar uno y no sé cuál me irá mejor para mi aplicación.



¿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.

En este módulo no entramos en los detalles de cómo se implementan internamente las listas, ni los árboles, ni los algoritmos de ordenación y búsqueda que se implementan para conseguir que el almacenamiento y las búsquedas sean más o menos eficientes, solo nos interesa tener unas nociones de las ventajas e inconvenientes que presentan unas estructuras frente a otras, para elegir las, y una vez elegidas, poder usarlas. Veamos un ejemplo de uso básico de la estructura `HashSet` y después, 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>();
```

Dado que `HashSet` es una implementación de la interfaz `Set`, podemos también crearlo de la siguiente forma:

```
Set<Integer> conjunto=new HashSet<Integer>();
```

En este segundo ejemplo simplemente se cambia el tipo usado para la variable, así la variable `conjunto` podrá apuntar a cualquier implementación de la interfaz `Set` (`HashSet`, `TreeSet`, etc.). ¡¡La potencia de usar interfaces en acción!!

Después podremos ir almacenando objetos dentro del conjunto usando el método `add()` definido por la interfaz `Set`, ya que extiende a `Collection`. 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("No se pudo añadir. El número "+n+" ya está en la lista.");
}
```

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

Autoevaluación

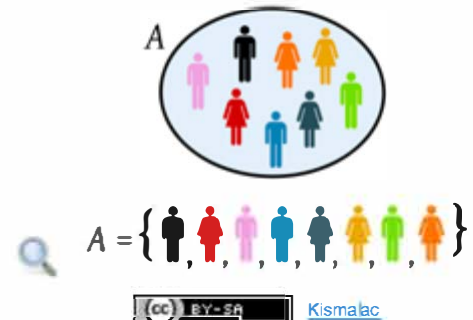
¿Cuál de las siguientes estructuras ordena automáticamente los elementos según su valor?

- ☐ HashSet.
- ☐ LinkedHashSet.
- ☐ TreeSet.

3.1.- Conjuntos (II). ¿Cómo acceder a sus elementos?

Y ahora te preguntarás, ¿cómo accedo a los elementos almacenados en un conjunto?

Para obtener todos los elementos almacenados en un conjunto hay que usar **iteradores**, que permiten obtener los elementos del conjunto uno a uno de forma secuencial (es la única forma de acceder a todos los elementos de un conjunto).



Los iteradores se verán en mayor profundidad más adelante, así que de momento los utilizaremos de forma "transparente" mediante un bucle **"for-each"** o bucle "para cada". Este tipo de bucles ya los vimos en la unidad dedicada a las estructuras de control, aunque apenas los habíamos utilizado hasta ahora. En el siguiente código tienes un ejemplo de uso de 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, 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 para ser procesado en cada iteración;
- ✓ colección es la colección en sí.

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

Ejercicio resuelto

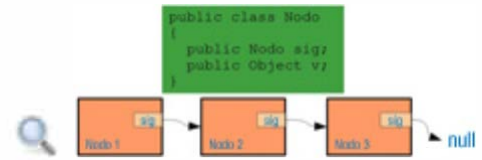
Realiza un pequeño programa que pregunte al usuario 5 números diferentes (almacenándolos en un HashSet), y que después calcule la suma de los mismos (usando un bucle **for-each**).

Mostrar retroalimentación

3.2.- Conjuntos (III). ¿En qué se diferencian LinkedHashSet y TreeSet?

¿En qué se diferencian las estructuras LinkedHashSet y TreeSet de la estructura HashSet?

Ya se comentó antes: la diferencia se encuentra básicamente en **su funcionamiento interno** que las hace más o menos eficientes y por tanto más o menos adecuadas **según el tipo de operaciones más frecuentes** que necesitemos realizar con los elementos que contienen.



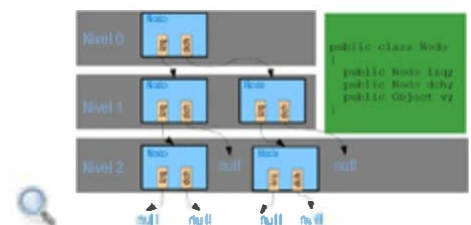
La estructura LinkedHashSet es una estructura que internamente funciona como una lista enlazada, aunque usa también **tablas hash** para poder acceder rápidamente a los elementos. Una lista enlazada es una estructura similar a la representada en la imagen de la derecha, la cual está compuesta por nodos (elementos que forman la lista) que van enlazándose entre sí. Un nodo contiene dos cosas: **el dato u objeto almacenado en la lista y una referencia al siguiente nodo de la lista**. Si no hay siguiente nodo, se indica poniendo nulo (null) en la referencia al 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 que nos proporciona Java podremos utilizar listas enlazadas sin tener que complicarnos, ni conocer, los detalles de su programación y funcionamiento interno, ya que nos las proporciona "listas para usar" cómodamente, con toda una serie de métodos disponibles, que nos proporciona su interfaz.

La estructura TreeSet, 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 figura de la derecha 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, éste 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**, como conjuntos que son, 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 TreeSet.	Conjunto LinkedHashSet.
Ejemplo de uso	<pre> TreeSet <Integer> t; t=new TreeSet<Integer>(); 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 <Integer> t; t=new LinkedHashSet<Integer>(); 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	<pre> 1 3 4 99 (el resultado sale ordenado por valor) </pre>	<pre> 4 3 1 99 (los valores salen ordenados según el momento de inserción en el conjunto) </pre>

En los ejemplos anteriores también se podría haber optado por usar una variable tipo Set. Por ejemplo, en el caso del TreeSet podría ser como sigue (con el mismo resultado):

```

Set <Integer> t;
t=new TreeSet<Integer>();

```


Autoevaluación

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

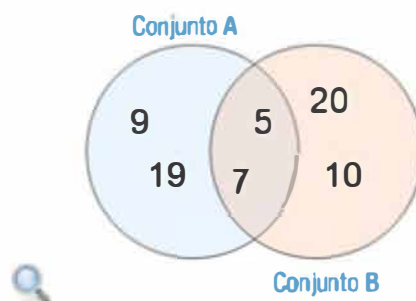
- ☐ Verdadero.
- ☐ Falso.

3.3.- Conjuntos (IV). Combinando datos de varias colecciones

¿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 de diferente tipo, cada una con 4 números enteros:

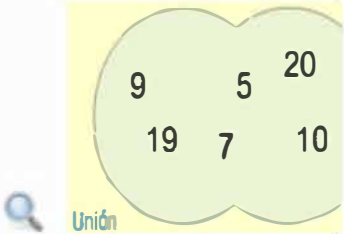
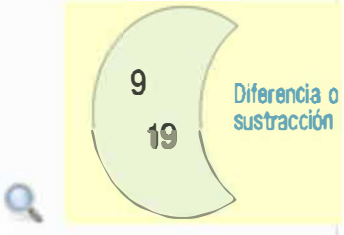
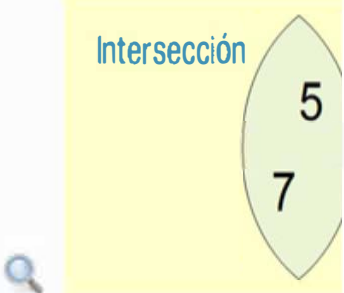


```
TreeSet<Integer> conjuntoA= new TreeSet<Integer>();  
conjuntoA.add(9); conjuntoA.add(19); conjuntoA.add(5); conjuntoA.add(7); // Elementos del conjunto A: 9, 19, 5 y 7
```

```
LinkedHashSet<Integer> conjuntoB= new LinkedHashSet<Integer>();  
conjuntoB.add(10); conjuntoB.add(20); conjuntoB.add(5); conjuntoB.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.	<pre><code>conjuntoA.addAll(conjuntoB)</pre>	<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> 
Diferencia. Eliminar los elementos del conjunto B que puedan estar en el conjunto A.	<pre><code>conjuntoA.removeAll(conjuntoB)</pre>	<p>Todos los elementos del conjunto A, que no estén en el conjunto B: 9, 19.</p> 
Intersección. Retiene los elementos comunes a ambos conjuntos.	<pre><code>conjuntoA.retainAll(conjuntoB)</pre>	<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**.

Para saber más

Puede que no recuerdes cómo era eso de los conjuntos, y dada la íntima relación de las colecciones con el álgebra de conjuntos, es recomendable que repases cómo era aquello, con el siguiente artículo de la **Wikipedia**.

 [Álgebra de conjuntos.](#)

Autoevaluación


Tienes un `HashSet` llamado `vocales` que contiene los elementos "a", "e", "i", "o", "u", y otro, llamado `vocalesFuertes` 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(vocalesFuertes);`
- ☐ `vocales.removeAll(vocalesFuertes);`
- ☐ No es posible hacer esto con `HashSet`, solo se puede hacer con `TreeSet` o `LinkedHashSet`.

3.4.- Conjuntos (V). Ordenando sus elementos

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 e implementar la interfaz Set, 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.



8
5
2
6
9
3
1
4
0
7

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 objeto1, Objeto objeto2) { ... }  
}
```

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

- ✓ Si el primer objeto (objeto1) se considera menor que el segundo (objeto2), debe devolver un número entero negativo.
- ✓ Si el primer objeto (objeto1) se considera mayor que el segundo (objeto2), debe devolver un número entero positivo.
- ✓ Si ambos son iguales, debe devolver 0.

Te suena, ¿verdad? Es el mismo criterio que seguía el método compareTo() que usábamos para comparar dos String... pero eso es otra historia

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, pensando en menor y mayor, puede ser un poco liosa, así que es recomendable en tales casos pensar en términos de delante y detrás, o de antes y después, más o menos de la siguiente forma:

- ✓ Si el primer objeto (objeto1) debe ir antes que el segundo objeto (objeto2), devolver entero negativo.
- ✓ Si el primer objeto (objeto1) debe ir después que el segundo objeto (objeto2), devolver entero positivo.
- ✓ Si ambos son iguales, debe devolver 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:

```
Set<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, éstos se tienen que ordenar de forma que la suma de sus atributos (a y b) sea descendente. ¿Cómo sería el comparador?

[Mostrar retroalimentación](#)

4.- Listas (I). Definición y diferencias con los conjuntos

Caso práctico



Juan se queda pensando después de que **Ana** le preguntara si sabía los tipos de colecciones que había en Java. Obviamente no lo sabía, son muchos tipos, pero ya tenía una respuesta preparada:

—Bueno, sea lo que sea, siempre puedes utilizar una lista para almacenar cualquier cosa. Yo siempre las uso, pues te permiten almacenar cualquier tipo de objeto, extraer uno de la lista sin tener que recorrerla entera, o buscar si hay o no un elemento en ella, de forma cómoda. Son para mí el mejor invento desde la rueda —dijo **Juan**.

—Ya, supongo, pero hay dos tipos de listas que me interesan, LinkedList y ArrayList. ¿Cuál es mejor? ¿Cuál me conviene más? —pregunta **Ana**.

¿En qué se diferencia una lista de un conjunto?

Analicemos cuáles son:

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:



[Peggy Marco \(Licencia Pixabay\)](#)

- ✓ Las listas **sí 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 objeto)`. El método `indexOf()` permite conocer la posición (**índice**) de un elemento (**objeto**), si dicho elemento no está en la lista retornará **-1**.
- ✓ `int lastIndexOf(Object objeto)`. El método `lastIndexOf()` nos permite obtener la última ocurrencia del objeto en la lista (dado que la lista sí 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 está en la posición de índice 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 `listaDeEnteros` es una lista de números enteros, ¿sería correcto poner "**`listaDeEnteros.add(listaDeEnteros.size(),3);`**"?

- ☐ Sí.
- ☐ No.

4.1.- Listas (II). ¿Cómo se usan?

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 cómo usar una `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:



[AllClear55](#) (Licencia Pixabay)

```
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 elemento de la lista.
int contador= 0;
for (Integer i: t) {
    contador++;
    System.out.println("Elemento " + contador + ": " + i); // Muestra cada elemento de la lista.
}
```

En el ejemplo anterior se realizan muchas operaciones, ¿pero 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> miLista=new ArrayList<Integer>(); // Declaración y creación del ArrayList de enteros.
miLista.add(10);
miLista.add(11); // Añadimos dos elementos a la lista.
miLista.set(miLista.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. Se generará un `ArrayList` que contendrá dos números, el 10 y el 12. Veamos ahora un ejemplo algo más difícil:

```
miLista.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.

las operaciones aplicadas a una sublista repercuten sobre la lista original
los elementos de la sublista, pero también se borrarán dichos elementos de la lista original:

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

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


Ten en cuenta que, al igual que pasaba con los conjuntos, las variables y atributos que contendrán una lista podrán crearse también de la siguiente forma:

```
List<Integer> miLista=new ArrayList<Integer>();
```

En vez de usar como tipo de variable `ArrayList`, `LinkedList`, etc. puedes usar como tipo `List`, dado que todas las anteriores serán implementaciones de dicha interfaz.

Para saber más

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

Dado el siguiente código, completa el cuadro vacío con el número que falta.

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

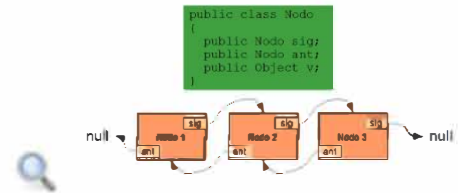
El valor de la variable suma después de ejecutarlo es

Enviar

4.2.- Listas (III). Diferencias entre LinkedList y ArrayList

¿Y en qué se diferencia un LinkedList de un ArrayList?

Las 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. Éstos se implementan utilizando arrays que se van redimensionando conforme se necesita más espacio o menos. La redimensión es transparente para nosotros, no nos enteramos cuando se produce, pero eso redundará 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?

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 pueden hacer aconsejable 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 primero que llega es el primero en ser atendido (FIFO, o Primero en Entrar, Primero en Salir). 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), devolverán true si se ha podido insertar el elemento al final de la LinkedList.
- ✓ E poll() devolverá 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() devolverá el primer elemento de la LinkedList pero no lo eliminará, permite examinarlo. Retornará null si la lista está vacía.

Las pilas, son todo lo contrario a las colas. Una pila es igual que una montaña de hojas en blanco (o una pila de platos): para añadir hojas nuevas (o platos), 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 (LIFO). 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).

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

- ☐ A.
- ☐ C.
- ☐ D.

4.3.- Listas (IV). ¿Es igual si los elementos son mutables o inmutables?

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 (`String`, `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.



[googlerankfaster](#) (Licencia Pixabay)

Imagínate la siguiente clase, que contiene un número:

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

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<>(); // Creamos una lista enlazada para objetos tipo Test.  
lista.add(p1); // Añadimos el primer objeto test.  
lista.add(p2); // Añadimos el segundo objeto test.  
for (Test p:lista){  
    System.out.println(p.numero); // Mostramos la lista de objetos.  
}
```

¿Qué mostraría por pantalla el código anterior? Simplemente mostraría los números 11 y 12 en líneas consecutivas.

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.numero=44;  
for (Test p:lista){  
    System.out.println(p.numero);  
}
```

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 o referencia a dicho objeto (solo hay una copia del objeto a la que se hace referencia desde distintos lugares).

Citas para pensar

“

Controlar la complejidad es la esencia de la programación.

Brian Kernighan

Autoevaluación

Los elementos de un ArrayList de objetos Short se copian al insertarse, ya que son objetos mutables.

- ☐ Verdadero.
- ☐ Falso.

5.- Conjuntos de pares clave/valor

Caso práctico



[Alexas Fotos \(Licencia Pixabay\)](#)

Juan se quedó pensativo después de la conversación con **Ana**. **Ana** se fue a su puesto a seguir trabajando, pero él se quedó dándole vueltas al asunto...

—Sí que está bien preparada **Ana**, me ha puesto en jaque y no sabía qué responder.


El hecho de no poder ayudar a **Ana** le frustró un poco.

De repente, apareció **María**. Entonces **Juan** aprovecha el momento para preguntar con más detalle acerca del trabajo de **Ana**. **María** se lo cuenta y de repente, se le enciende una bombilla

a **Juan**:

—Vale, creo que puedo ayudar a **Ana** en algo, le aconsejaré usar **mapas** y le explicaré cómo se usan.

¿Cómo almacenarías los datos de un diccionario?

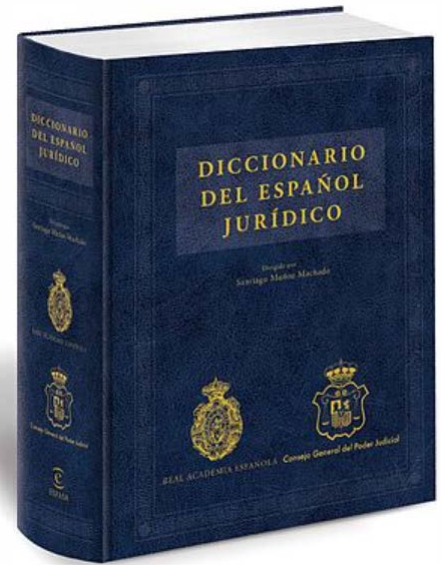
Tenemos por un lado cada palabra y por otro su significado. Para resolver este problema existe precisamente un tipo de  array asociativo: los **mapas o diccionarios**, que permiten almacenar pares de valores conocidos como **pares clave-valor**. La clave se utiliza para acceder al valor, como una entrada 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`
- ✓ `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.

[Real Academia Española. \(CC BY-SA\)](#)



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 devolverá el valor asociado que tenía antes, si la llave no existía, entonces devolverá null.
V get(Object key)	Obtiene el valor asociado a una llave ya almacenada en el mapa. Si no existe la llave, devolverá null.
V remove(Object key)	Elimina la llave y el valor asociado. Devuelve el valor asociado a la llave, por si lo queremos utilizar para algo, o null, si la llave no existe.
boolean containsKey(Object key)	Devolverá true si el mapa tiene almacenada la llave pasada por parámetro, false en cualquier otro caso.
boolean containsValue(Object value)	Devolverá true si el mapa tiene almacenado el valor pasado por parámetro, false en cualquier otro caso.
int size()	Devolverá el número de pares llave y valor almacenado en el mapa.
boolean isEmpty()	Devolverá true si el mapa está vacío, false en cualquier otro caso.
void clear()	Vacía el mapa.

Al igual que pasa con las colecciones anteriores (listas y conjuntos), puedes crear un mapa de la siguiente forma:

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

En vez de usar `HashMap` como tipo para la variable (`HashMap miMapa` por ejemplo), puedes usar `Map` (por ejemplo: `Map miMapa`). Esto puedes hacerlo porque `HashMap` es una implementación de la interfaz `Map`, y lo mismo es aplicable para el resto de implementaciones de mapa: `LinkedHashMap`, `TreeMap`, etc.

Autoevaluación

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

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

Enviar

6.- Iteradores (I). ¿Qué son y cómo se usan?

Caso práctico

Juan se acercó a la mesa de **Ana** y le dijo:

—**María** me ha contado la tarea que te ha encomendado y he pensado que quizás te convendría usar mapas en algunos casos. Por ejemplo, para almacenar los datos del pedido asociados con una etiqueta: nombre, dirección, fecha, etc. Así creo que te será más fácil encontrar la información de un pedido concreto (valor) asociado a esa etiqueta (clave, o llave).



[stevepb](#) (Licencia Pixabay)

—La verdad es que pensaba almacenar los datos del pedido en una clase especial llamada Pedido. No tengo ni idea de qué son los mapas —dijo **Ana**—, supongo que son como las listas. ¿Tienen iteradores?

—Según me ha contado **María**, no necesitas hacer tanto, no es necesario crear una clase específica para los pedidos. Y respondiendo a tu pregunta, los mapas no tienen iteradores, pero hay una solución... Te explico.



[gerall \(Licencia Pixabay\)](#)

¿Qué son los iteradores realmente?

Son un mecanismo que nos permite recorrer todos los elementos de una colección de forma secuencial, sencilla y 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().** Devolverá true si le quedan más elementos a la colección por visitar. false en caso contrario.
- ✓ **E next().** Devolverá 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 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 numero=it.next(); // Escogemos el siguiente elemento.  
    if (numero%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 ti 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())`?

6.1.- Iteradores (II). ¿Qué características debemos considerar?

¿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. Se eliminan de la lista aquellos que son pares y sólo 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 que al no especificar el tipo de objeto del iterador, tenemos que usar la conversión explícita de tipos (**casting**) en la línea 7.

Comparación de usos de los iteradores, con o sin conversión de tipos.

Ejemplo indicando el tipo de objeto de iterador	Ejemplo no indicando el tipo de objeto del iterador
<pre>ArrayList<Integer> lista=new ArrayList<>(); for (int i=0;i<=10;i++){ lista.add(i); } Iterator<Integer> it=lista.iterator(); while (it.hasNext()) { Integer t=it.next(); if (t%2==0) it.remove(); }</pre>	<pre>ArrayList<Integer> lista=new ArrayList<>(); for (int i=0;i<=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 incómoda excepción. Usar genéricos aporta grandes ventajas, pero usándolos adecuadamente.



[TheDigitalArtist \(Licencia Pixabay\)](#)

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 cómo sería para el segundo caso, el más sencillo:

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

Reflexiona

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 por qué se pueden producir dichos problemas?

Mostrar retroalimentación

Autoevaluación

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

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

7.- Algoritmos (I). ¿Qué podemos hacer con las colecciones?

Caso práctico

Ada se acercó a preguntar a **Ana** cómo llevaba la tarea que le había encomendado **María**. **Ada** era la jefa y **Ana** le tenía mucho respeto. Era una tarea importante, así que prestó mucha atención.

Ana le enseñó el código que estaba elaborando, le dijo que en un principio había pensado crear una clase llamada Pedido para almacenar los datos del pedido, pero que **Juan** le recomendó usar mapas para almacenar los pares de valor y dato. Así que se decantó por usar mapas para ese caso. Le comentó también que para almacenar los artículos sí había creado una pequeña clase llamada Artículo. **Ada** le dio el visto bueno:



—Pues **Juan** te ha recomendado de forma adecuada, no vas a necesitar hacer ningún procesamiento especial de los datos del pedido, solo convertirlos de un formato específico. Eso sí, sería recomendable que los artículos del pedido vayan ordenados por código de artículo —dijo **Ada**.

—¿Ordenar los artículos? Vaya, que jaleo —respondió **Ana**.

—Arriba ese ánimo mujer, si has usado listas es muy fácil, déjame ver tu código y te explicaré cómo hacerlo.

Ejercicio Resuelto

A continuación te mostramos el código del ejemplo que procesaría los datos de un pedido que se recibe en un "archivo de texto" (simulado por un array de String) guardándolos en un Map.

Primero te mostramos el código del ejemplo, pulsando el botón de "Mostrar retroalimentación".

Mostrar retroalimentación

Y a continuación, la salida que produciría.

Mostrar retroalimentación

L

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.**
- ✓ **Partir cadenas** y almacenar el resultado en un array.



[geralt \(Licencia Pixabay\)](#)

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 cómo ordenarlos. Como se explicó 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 con números
<pre>Integer[] array={10,9,99,3,5}; Arrays.sort(array);</pre>	<pre>ArrayList<Integer> lista=new ArrayList<>(); lista.add(10); lista.add(9); lista.add(99); lista.add(3); lista.add(5); Collections.sort(lista);</pre>

7.1.- Algoritmos (II). Ordenando los elementos de la colección

En Java hay dos mecanismos para cambiar la forma en la que los elementos se ordenan. ¿Recuerdas la tarea que **Ada** pidió a **Ana**?

Así es, **Ada** pidió a **Ana** que los artículos del pedido aparecieran ordenados por código de artículo. Imagina que tienes los artículos 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 Artículo {  
    public String codigoArticulo; // Código de artículo  
    public String descripcion; // Descripción del artículo.  
    public int cantidad; // Cantidad a proveer del artículo.  
}
```



[geralt \(Licencia Pixabay\)](#)

La **primera forma** de ordenar consiste en crear una clase que implemente la interfaz `java.util.Comparator`, lo que supone implementar 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 articulo1, Articulo articulo2) {  
        return articulo1.codigoArticulo.compareTo(articulo2.codigoArticulo);  
    }  
}
```

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 codigoArticulo;
    public String descripcion;
    public int cantidad;

    @Override
    public int compareTo(Articulo articulo) {
        return codigoArticulo.compareTo(articulo.codigoArticulo);
    }
}

```

Del ejemplo anterior se pueden resaltar 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 Articulo 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 el objeto que se pasa por parámetro es igual al objeto ha llamado al método, se tendría que devolver 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.

7.2.- Algoritmos (III). Algunas operaciones adicionales con colecciones

¿Qué más ofrecen las clases `java.util.Collections` y `java.util.Arrays` de Java?

Una vez vista la ordenación, que quizás es 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
Desordenar una lista.	Desordena una lista, este método no está disponible para arrays.	<code>Collections.shuffle (lista);</code>
Rellenar una lista o array.	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>
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.	<code>Collections.binarySearch(lista,elemento);</code> <code>Arrays.binarySearch(array, elemento);</code>
Convertir un array a lista.	Permite rápidamente convertir un array a una lista de elementos, y es 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=Arrays.asList(array);</code> Si el tipo de dato almacenado en el array es conocido (<code>Integer</code> por ejemplo), es conveniente especificar el tipo de objeto de la lista: <code>List<Integer>lista = Arrays.asList(array);</code>
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 <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>
Dar la vuelta.	Da la vuelta a una lista, poniéndola en orden inverso al que tiene.	<code>Collections.reverse(lista);</code>

Otra operación que no se ha visto hasta ahora es la **dividir una cadena en partes**.

Cuando una cadena está formada internamente por trozos de texto claramente delimitados por un separador (una coma, un punto y coma o cualquier otro), es posible dividir la cadena y obtener cada uno de los trozos de texto por separado en un array de cadenas. Es una operación sencilla, pero requiere conocer el funcionamiento de los arrays y de las expresiones regulares para su uso. Para poder realizar esta operación, usaremos el método

split() de la clase String. El delimitador o separador es una expresión regular, único argumento del método split(), y puede ser obviamente todo lo complejo que sea necesario:



[pexels](#) (Dominio público)

```
String texto="Z,B,A,X,M,O,P,U";  
String[] partes=texto.split(",");  
Arrays.sort(partes);
```

En el ejemplo anterior la cadena texto contiene una serie de letras separadas por comas. La cadena se ha dividido con el método split(), y se ha guardado cada carácter por separado en un array. Después se ha ordenado el array.

¡Increíble lo que se puede llegar a hacer con solo tres líneas de código!

Para saber más

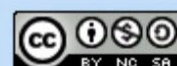
En el siguiente vídeo podrás ver en qué consiste la búsqueda binaria y cómo se aplica de forma sencilla:

[Resumen textual alternativo para "Método de búsqueda binaria"](#)

Ahora bien, este vídeo contiene un par de erratas. A ver si eres capaz de encontrarlas. Te damos una pista: la primera se encuentra sobre el minuto y medio, y la segunda sobre los dos minutos y medio.

Condiciones y términos de uso de los materiales

Materiales desarrollados inicialmente por el Ministerio de Educación, Cultura y Deporte y actualizados por el profesorado de la Junta de Andalucía bajo licencia Creative Commons **BY-NC-SA**.



Antes de cualquier uso leer detenidamente el siguiente [Aviso legal](#)

Historial de actualizaciones

Versión: 02.00.02		Fecha de actualización: 14/02/22
Actualización de materiales y correcciones menores.		
Versión: 02.00.00	Fecha de actualización: 08/06/21	Autoría: Diosdado Sánchez Hernández
<p>Ubicación: 1.- Introducción.</p> <p>Mejora (tipo 1): En el Para saber más hay un error ortográfico: Donde dice Verás que es un basto tema lo que abarca. Debe decir Verás que es un vasto tema lo que abarca.</p> <p>Ubicación: Toda la unidad</p> <p>Mejora (tipo 3): Cambiar de orden esta unidad. Esta pasaría a ser la 8, y la de estructuras de almacenamiento, la 7.</p> <p>En cualquier curso sobre introducción a la Programación se estudian siempre primero las estructuras de datos internas (tanto estáticas: arrays, cadenas, etc. como dinámicas: listas, tablas hash, árboles, etc.) y posteriormente las estructuras de datos externas (archivos y sus diversas variedades). Se considera lo más didáctico y además mucho más práctico para posteriormente poder realizar ejercicios y tareas que permitan almacenar en archivos el contenido de estructuras dinámicas. Si no, aún no se dispone de estructuras lo suficientemente complejas (salvo arrays) que justifiquen la necesidad de estructuras de almacenamiento externas.</p> <p>Implicaría revisar orientaciones para el alumnado.</p> <p>Ubicación: Número de unidad</p> <p>Mejora (Mapa conceptual): Modificados los nombres de enlaces y archivos donde esté involucrada el número de la unidad</p> <p>Ubicación: Cabecera y descripción</p> <p>Mejora (Orientaciones del alumnado): Modificado el nombre y número de la unidad</p>		

Versión: 01.03.00	Fecha de actualización: 09/03/21	Autoría: Juan Antonio Reina Gómez
Ubicación: Apartados 5.2.1 y 5.5 Mejora (tipo 2): Cambio de recurso Flash Player por fichero .PDF Ubicación: Apartado 7.3 Mejora (tipo 1): Link de un recurso web roto (se actualiza el link por uno correcto) Ubicación: Apartado 7 Mejora (tipo 1): Link de la licencia de una imagen roto (se actualiza por uno correcto) Ubicación: Apartado 3 Mejora (tipo 1): Link de la licencia de una imagen roto (se actualiza el link por uno correcto)		
Versión: 01.02.00	Fecha de actualización: 16/12/19	Autoría: José Javier Bermúdez Hernández
Ubicación: XML Mejora (tipo 1): Se ha movido la parte de XML que estaba en el tema 8 a este tema. Ubicación: Cmabio de orden de unidad Mejora (tipo 2): SE trata de cambiar la secuenciación, de modo que la unidad 7 pasa a ser la 8 y la 8 para a ser la 7. Ubicación: Mapa Mejora (Mapa conceptual): Añadida rama XML Ubicación: Índice Mejora (Orientaciones del alumnado): Se ha modificado el índice para añadir el último apartado que no existía, sobre XML		
Versión: 01.01.00	Fecha de actualización: 05/11/15	Autoría: José Javier Bermúdez Hernández
Ubicación: Sección 4.1 Mejora (tipo 2): Añadir un Debes conocer con el vídeo: https://www.youtube.com/watch?v=vTn_mXIA9w4 Ubicación: Sección 4.2 Mejora (tipo 2): Eliminar el Debes conocer que hay, pues trata sobre excepciones que ya se han visto en temas anteriores, y añadir un Para Saber más a: http://www.dit.upm.es/~pepe/libros/vademecum/index.html?n=375.html Ubicación: Anexo de licencias Mejora (tipo 1): Eliminar el anexo de licencias, poniendo las credenciales al pie de la propia imagen Ubicación: No especificada. Mejora (Examen online): Revisar la pregunta: Cuando se escribe en un fichero secuencial hay que tener la precaución de ir escribiendo las cadenas de caracteres con el mismo tamaño, de manera que sepamos luego el tamaño del registro que tenemos que leer.		
Versión: 01.00.00	Fecha de actualización: 11/02/14	
Versión inicial de los materiales.		