

TEMA 8

INDICE

1.- Introducción a las estructuras de almacenamiento.....	4
2.- Cadenas de caracteres.	5
2.1.- Operaciones avanzadas con cadenas de caracteres. (I).....	6
2.1.1.- Operaciones avanzadas con cadenas de caracteres. (II)	7
2.1.2.- Operaciones avanzadas con cadenas de caracteres. (III)	8
Sintaxis de las cadenas de formato y uso del método format.....	11
2.1.3.- Operaciones avanzadas con cadenas de caracteres. (III)	13
2.1.4.- Operaciones avanzadas con cadenas de caracteres. (IV)	14
2.2.- Expresiones regulares. (I).....	15
2.2.1.- Expresiones regulares. (II)	17
2.2.2.- Expresiones regulares. (III)	18
3.- Creación de arrays.....	20
3.1.- Uso de arrays unidimensionales.	22
3.2.- Inicialización.....	23
4.- Arrays multidimensionales.....	25
4.1.- Uso de arrays multidimensionales.....	26
4.2.- Inicialización de arrays multidimensionales.	27
5.- Clases y métodos genéricos. (I)	28
5.1.- Clases y métodos genéricos. (II).....	29
6.- Introducción a las colecciones. (I)	32
7.- Conjuntos. (I)	34
7.1.- Conjuntos. (II).....	35
Ejercicio resuelto	35
7.2.- Conjuntos. (III).....	36
7.3.- Conjuntos. (IV)	37
7.4.- Conjuntos. (V)	38
Ejercicio resuelto	38
8.- Listas. (I)	40
8.1.- Listas. (II)	41
8.2.- Listas. (III)	42
8.3.- Listas. (IV)	43
9.-Conjuntos de pares clave/valor.	44
10.- Iteradores. (I)	45
10.1.- Iteradores. (II)	45
11.- Algoritmos. (I).....	47
11.1.- Algoritmos. (II)	50
11.2.- Algoritmos. (III)	51
12.- Tratamiento de documentos estructurados XML.....	53
12.1.- ¿Qué es un documento XML?.....	54
12.2.- Librerías para procesar documentos XML. (I)	55
12.2.1.- Librerías para procesar documentos XML. (II).....	56
12.3.-Manipulación de documentos XML.(I).....	59
12.3.1.-Manipulación de documentos XML.(II)	60
Obtener la lista de hijos de un elemento y procesarla.....	60
Añadir un nuevo elemento hijo a otro elemento.....	60
12.3.2.-Manipulación de documentos XML.(III)	61
Eliminar un elemento hijo de otro elemento.	61
Cambiar el contenido de un elemento cuando solo es texto.....	61
12.3.3.-Manipulación de documentos XML.(IV)	62
Atributos de un elemento.....	65

5.- 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 para luego hacer el volcado XML, pero no se como 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 los artículos del pedido, y no sé como 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.

¿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 de programación. 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 como transformar un método normal en genérico:

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 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 **a** es mayor, y un número menor de cero si el array **b** es mayor, pero uno es genérico y el otro no. La versión genérica del módulo 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.

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>

..

5.1.- 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:

```
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.

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

Dos o más parámetros de tipo. (I)

```
public class util {
    public static <T,M> int sumaDeLongitudes (T[]
a, M[] b)
    {
        return a.length+b.length;
    }
}
```

Si un método genérico necesita tener dos o más parámetros genéricos, podemos indicarlo separándolos por comas. En el ejemplo anterior se suman las longitudes de dos arrays que no tienen que ser del mismo tipo.

Dos o más parámetros de tipo. (II)

```
class terna <A,B,C> {
    A a; B b; C c;
    public terna (A a, B b, C c)
    {this.a=a; this.b=b; this.c=c;}
    public A getA () { return a; }
    public B getB () { return b; }
    public C getC () { return c; }
}
```

Si una clase genérica necesita tener dos o más parámetros genéricos, podemos indicarlo separándolos por comas. En el ejemplo anterior se muestra una clase que almacena una terna de elementos de diferente tipo base que están relacionados entre sí.

Dos o más parámetros de tipo. (II)

```
Integer[] a1={0,1,2,3,4};
Double[] a2={0d,1d,2d,3d,4d};
util.<Integer,Double>sumaDeLongitudes(a1,a2);
```

Usar un método o una clase con dos o más parámetros genéricos es sencillo, a la hora de invocar el método o crear la clase, se indican los tipos base separados por coma.

Métodos con tipos adicionales

```
class Util <A> {
    A a;
    Util (A a) { this.a=a; }
    public <B> void Salida (B b) {
        System.out.println(a.toString() +
        b.toString());
    }
}
```

Una clase genérica puede tener unos parámetros genéricos, pero si en uno de sus métodos necesitamos otros parámetros genéricos distintos, no hay problema, podemos combinarlos.

Inferencia de tipos. (I)

```
Integer[] a1={0,1,2,3,4};
Double[] a2={0d,1d,2d,3d,4d};
util.<Integer,Double>sumaDeLongitudes(a1,a2);
util.sumaDeLongitudes(a1,a2);
```

No siempre es necesario indicar los tipos a la hora de instanciar un método genérico. A partir de Java 7, es capaz de determinar los tipos a partir de los parámetros. Las dos expresiones de arriba sería válidas y funcionarían. Si no es capaz de inferirlos, nos dará un error a la hora de compilar.

Inferencia de tipos. (II)

```
Integer a1=0; Double d1=1.3d; Float f1=1.4f;
Terna <Integer,Double,Float> t= new
Terna<>(a1,d1,f1);
```

A partir de Java 7 es posible usar el operador diamante (“<>”) para simplificar la intanciación o creación de nuevos objetos a partir de clases genéricas. Cuidado, esto es solo a partir de Java 7.

Limitación de tipos

```
public class Util {
    public static <T extends Number> Double Sumar
    (T t1, T t2){
        return new Double(t1.doubleValue() +
        t2.doubleValue());
    }
}
```

Se pueden limitar el conjunto de tipos que se pueden usar con una clase o método genérico usando el operador “extends”. El operador **extends** permite indicar que la clase que se pasa como parámetro genérico tiene que derivar de una clase específica. En el ejemplo, no se admitirá ninguna clase que no derive de **Number**, pudiendo así realizar operaciones matemáticas.

Paso de clases genéricas por parámetro

```
public class Ejemplo <A> {
    public A a;
}
...
void test (Ejemplo<Integer> e) {...}
```

Cuando un método tiene como parámetro una clase genérica (como es el caso del método **test** del ejemplo), se puede especificar cual debe ser el tipo base usado en la instancia de la clase genérica que se le pasa como argumento. Esto sirve permite, entre otras cosas, crear diferentes versiones de un mismo método (sobrecarga), dependiendo del tipo base usado en la instancia de la clase genérica se ejecutará una versión u otra.

Paso de clases genéricas por parámetro. Wildcards. (I)

```
public class Ejemplo <A> {
    public A a;
}
...
void test (Ejemplo<?> e) {...}
```

Cuando un método admite como parámetro una clase genérica, en la que no importa el tipo de objeto sobre la que se ha creado, podemos usar el interrogante para indicar “cualquier tipo”.

Paso de clases genéricas por parámetro. Wildcards. (II)

```
public class Ejemplo <A> {
    public A a;
}
...
void test (Ejemplo<? extends Number> e) {...}
```

También es posible limitar el conjunto de tipos que una clase genérica puede usar, a través del operador **extends**. El ejemplo anterior es como decir “cualquier tipo que derive de **Number**”.

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);

6.- Introducción a las colecciones. (I)

Caso práctico.

A **Ana** las listas siempre se le han atragantado, por eso no las usa. 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, ha concluido 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.

¿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. En algunos lenguajes de programación su uso es algo más complejo (como es el caso de C++), pero en Java su uso es bastante sencillo, es algo que descubrirás a lo largo de lo que queda de tema.

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

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

- ✓ Método `void clear()`: vaciar la colección.

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

7.- Conjuntos. (I)

Caso práctico.

Ana se toma un descanso, se levanta 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:

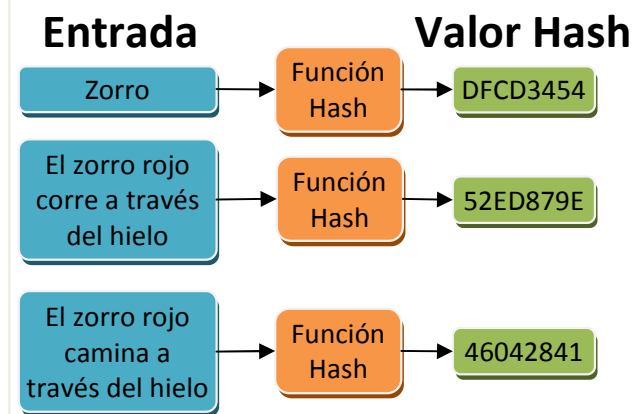
-¿Cuántos tipos de colecciones hay? ¿Tú te los sabes? -pregunta Ana.

-¿Yo? ¡Qué va! Normalmente consulto la documentación cuando los 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.

¿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` (estructura de datos formada básicamente por un array donde la posición de los datos va determinada por una función hash, permitiendo localizar la información de forma extraordinariamente rápida. Los datos están ordenados en la tabla en base a un resumen numérico de los mismos (en hexadecimal generalmente) obtenido a partir de un algoritmo para cálculo de resúmenes, denominadas funciones hash. El resumen no tiene significado para un ser humano, se trata simplemente de un mecanismo para obtener un número asociado a un conjunto de datos. El inconveniente de estas tablas es que los datos se ordenan por el resumen obtenido, y no por el valor almacenado. El resumen, de un buen algoritmo hash, no se parece en nada al contenido almacenado) 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 (estructura de datos que almacena los objetos enlazándolos entre sí a través de un apuntador de memoria o puntero, manteniendo un orden, que generalmente es el de momento de inserción, pero que puede ser otro. Cada dato se almacena en una estructura llamada nodo en la que existe un campo, generalmente llamado siguiente, que contiene la dirección de memoria del siguiente nodo (con el siguiente dato)) 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.

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*.

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



HashSet



LinkedHashSet



TreeSet

7.1.- Conjuntos. (II).

Y ahora te preguntará, ¿cómo accedo a los elementos almacenados en un conjunto? Para obtener los elementos almacenados en un conjunto hay que usar iteradores, que permiten obtener los elementos del conjunto uno a uno de forma secuencial (*no hay otra forma de acceder a los elementos de un conjunto, es su inconveniente*). Los iteradores se ven en mayor profundidad más adelante, de momento, vamos a usar iteradores de forma transparente, a través de una estructura `for` especial, denominada bucle “`for-each`” o bucle “para cada”. En el siguiente código se usa un bucle *foreach*, 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`).

Respuesta

Una solución posible podría ser la siguiente. Para preguntar al usuario un número y para mostrarle la información se ha usado la clase `JOptionPane`, pero podrías haber utilizado cualquier otro sistema. Fijate en la solución y verás que el uso de conjuntos ha simplificado enormemente el ejercicio, permitiendo al programador o la programadora centrarse en otros aspectos:

```
import java.util.HashSet;
import javax.swing.JOptionPane;

public class ejemplo {
    public static void main(String[] args)
    {
        HashSet<Integer> conjunto=new HashSet<Integer>();
        String str;
        do {
            str=JOptionPane.showInputDialog("Introduce un número "+(conjunto.size()+1)+":");
            try {
                Integer n= Integer.parseInt(str);
                if (!conjunto.add(n))
                    JOptionPane.showMessageDialog(null, "Número ya en la lista. Debes introducir otro.");
            }
            catch (NumberFormatException e)
            { JOptionPane.showMessageDialog(null,"Número erróneo."); }
        }
    }
}
```

```

    } while (conjunto.size() < 5);

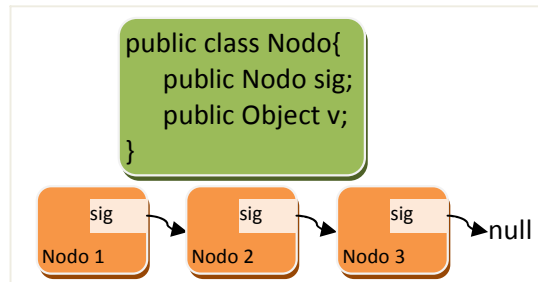
    // Calcular la suma
    Integer suma = new Integer(0);
    for (Integer i: conjunto) {
        suma = suma + i;
    }
    JOptionPane.showMessageDialog(null, "La suma es: " + suma);
}
}

```

7.2.- Conjuntos. (III).

¿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 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 el siguiente nodo de la lista. Si no hay siguiente nodo, se indica poniendo nulo (*null*) en la variable que contiene el siguiente nodo.

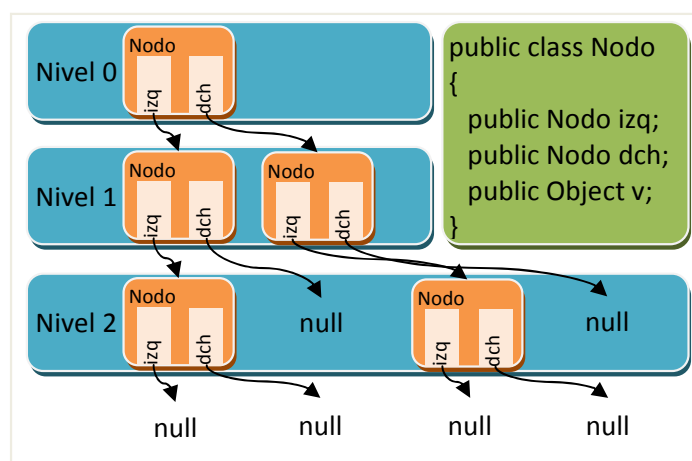


Las listas enlazadas tienen un montón de operaciones asociadas en las que no vamos a profundizar: eliminación de un nodo de la lista, inserción de un nodo al final, al principio o entre dos nodos, etc. Gracias a las colecciones podremos utilizar listas enlazadas sin tener que complicarnos en detalles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

	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>

Un árbol cuyos nodos solo pueden tener un único nodo hijo, en realidad es una lista.



Verdadero



Falso

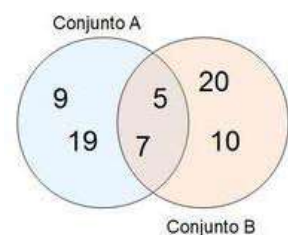
7.3.- Conjuntos. (IV)

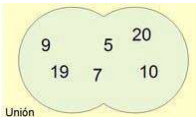
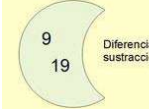

¿Cómo podría copiar los elementos de un conjunto de uno a otro? ¿Hay que usar un bucle for y recorrer toda la lista para ello? ¡Qué va! Para facilitar esta tarea, los conjuntos, y las colecciones en general, facilitan un montón de operaciones para poder combinar los datos de varias colecciones. Ya se 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> 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:






Combinación.	Código.	Elementos finales del conjunto A.
Unión. Añadir todos los elementos del conjunto B en el conjunto A.	<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. 
Diferencia. Eliminar los elementos del conjunto B que puedan estar en el conjunto A.	<code>A.removeAll(B)</code>	Todos los elementos del conjunto A, que no estén en el conjunto B: 9, 19. 
Intersección. Retiene los elementos comunes a ambos conjuntos.	<code>A.retainAll(B)</code>	Todos los elementos del conjunto A, que también están en el conjunto B: 5 y 7. 

Recuerda, estas operaciones son comunes a todas las colecciones.

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.

http://es.wikipedia.org/wiki/%C3%81lgebra_de_conjuntos

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.

7.4.- Conjuntos. (V)

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

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, ¿cómo sería el comparador?

Respuesta

Una de las posibles soluciones a este problema podría ser la siguiente:

```
class ComparadorDeObjetos implements Comparador<Objeto> {  
    @Override  
    public int compare(Objeto o1, Objeto o2) {  
        int sumao1=o1.a+o1.b;  int sumao2=o2.a+o2.b;  
        if (sumao1<sumao2) return 1;  
        else if (sumao1>sumao2) return -1;  
        else return 0;  
    }  
}
```

8.- Listas. (I)

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 lo que sea. Yo siempre las uso, pues te permiten almacenar cualquier tipo de objeto, extraer uno de las lista sin tener que recorrerla entera, 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? -respondió Ana.

¿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.

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



Sí



No

Inserta un elemento al final de la lista y es equivalente a poner `M.add(3)`.

8.1.- Listas. (II)

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

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

```
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 sublista y en la lista original.

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.

http://es.wikipedia.org/wiki/Lista_enlazada

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()+1); Integer suma = t.get(0) + t.get(1);
```

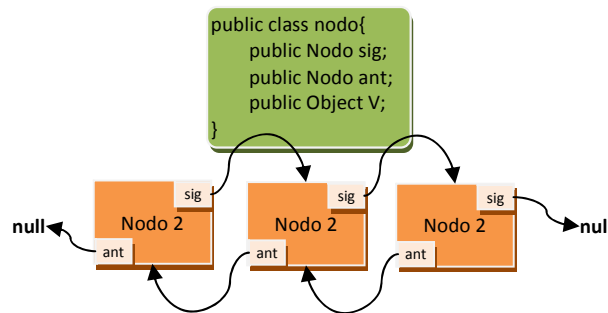
El valor de la variable suma después de ejecutarlo es

3

8.2.- Listas. (III)

¿Y en qué se diferencia un `LinkedList` de un `ArrayList`? Los `LinkedList` utilizan listas doblemente enlazadas, que son listas enlazadas (como se vio en un apartado anterior), pero que permiten ir hacia atrás en la lista de elementos. Los elementos de la lista se encapsulan en los llamados nodos.

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



No es el caso de los `ArrayList`. Estos se implementan utilizando arrays que se van redimensionando conforme se necesita más espacio o menos. La redimensión es transparente a nosotros, no nos enteramos cuando se produce, pero eso 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? **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 que llega es el primero en ser atendido (FIFO en inglés). 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.

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

8.3.- Listas. (IV)

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).

"Controlar la complejidad es la esencia de la programación."

Brian Kerniga

En el siguiente documento encontrarás mucha información adicional sobre las estructuras de datos, incluyendo algunas estructuras de datos adicionales menos usadas que no se abordan en este tema. Algunas palabras cambian, por ejemplo, llama "arreglos" a los "arrays" (ya se comentó con anterioridad la existencia de esa otra nomenclatura). Es un texto que profundiza bastante, en las estructuras de datos, así que léelo con paciencia y sin prisas:

http://www.utim.edu.mx/~svalero/docs/ED_Java.pdf

Los elementos de un ArrayList de objetos Short se copian al insertarse al ser objetos mutables.



Verdadero
Falso

Los elementos se pasan por copia por ser inmutables, no mutables.

9.-Conjuntos de pares clave/valor.

Caso práctico.

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... “Si 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**... dice: “Vale, creo que puedo ayudar a **Ana** en algo, le aconsejare usar mapas y le explicaré como se usan.”

¿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 si. 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:

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

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

```
HashMap<String,String> datos=new HashMap<String,String>();
datos.put("A","44");
System.out.println(Integer.parseInt(datos.get("A"))-4);
```

10.- Iteradores. (I)

Caso práctico.

Juan se acerca 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 generar luego el XML.

-La verdad es que pensaba almacenar los datos del pedido en una clase especial llamada Pedido. No tengo ni idea de que 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.

¿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();
```

Fijate que se ha especificado un parámetro para el tipo de dato genérico en el iterador (poniendo "`<Integer>`" después de `Iterator`). Esto es porque los iteradores son también clases genéricas, y es necesario especificar el tipo base que contendrá el iterador. Sino se especifica el tipo base del iterador, igualmente nos permitiría recorrer la colección, pero retornará objetos tipo `Object` (clase de la que derivan todas las clases), con lo que nos veremos obligados a forzar la conversión de tipo.

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

- ✓ `boolean hasNext()`. Retornará `true` si le quedan más elementos a la colección por visitar. `False` en caso contrario.
- ✓ `E next()`. Retornará el siguiente elemento de la colección, si no existe siguiente elemento, lanzará una excepción (`NoSuchElementException` para ser exactos), con lo que conviene chequear primero si el siguiente elemento existe.
- ✓ `remove()`. Elimina de la colección el último elemento retornado en la última invocación de `next` (no es necesario pasárselo por parámetro). Cuidado, si `next` no ha sido invocado todavía, saltará una incomoda excepción.

¿Cómo recorreríamos una colección con estos métodos? Pues de una forma muy sencilla, un simple bucle mientras (`while`) con la condición `hasNext()` nos permite hacerlo:

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

¿Qué elementos contendría la lista después de ejecutar el bucle? Efectivamente, solo los números impares.

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())`"?

10.1.- 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 6.

Ejemplo indicando el tipo de objeto de iterador.	Ejemplo no indicando el tipo de objeto del iterador,
<pre>ArrayList<Integer> lista=new ArrayList<Integer>(); 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<Integer>(); 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 esta 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.

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:

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

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

Si usas iteradores, y piensas eliminar elementos de la colección (e incluso de un mapa), debes usar el método `remove` del iterador y no el de la colección. Si eliminas los elementos utilizando el método `remove` de la colección, mientras estás dentro de un bucle de iteración, o dentro de un bucle `for-each`, los fallos que pueden producirse en tu programa son impredecibles. ¿Logras adivinar porqué se pueden producir dichos problemas?

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

¿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

11.- Algoritmos. (I)

Caso práctico.

Ada se acercó a preguntar a **Ana**. **Ada** era la jefa y **Ana** le tenía mucho respeto. **Ada** le preguntó cómo llevaba la tarea que le había encomendado **María**. 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 si había creado una pequeña clase llamada *Articulo*. **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 a XML. 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.

-Si, aquí está, es el siguiente, espero que te guste:

```
import java.io.BufferedReader;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.Scanner;
import java.util.logging.Level;
import java.util.logging.Logger;
import java.util.regex.Matcher;
import java.util.regex.Pattern;

/*
 * Clase destinada a almacenar los datos de un artículo.
 */
class Articulo {
    public String codArticulo;
    public String descripcion;
    public int cantidad;
}

/* Clase que se encarga de procesar un pedido.
 * El archivo con el pedido debe estar en codificación UTF8, sino no funciona
 * bien.
 * Tu sistema debe soportar de forma nativa la coficación UTF8, sino no
 * funcionará del todo bien.
 */
public class ProcesarArchivo2 {

    /* Entrada contendrá una instancia de la clase Scanner que permitirá
    leer las teclas pulsadas desde teclado */
    static Scanner entrada = new Scanner(System.in);
    /* Definimos las expresiones regulares que usaremos una y otra vez para
    cada línea del pedido. La expresión regular "seccion" permite detectar
    si hay un comienzo o fin de pedido, y la expresión campo, permite detectar
    si hay un campo con información del pedido. */
    static Pattern seccion = Pattern.compile("^#[ ]*(FIN)?[ ]*(PEDIDO|ARTICULOS)[ ]*##$");
    static Pattern campo = Pattern.compile("^(.+):.*\\{(.*?)\\}$");
    static Pattern articulo = Pattern.compile ("^\\{(.*?)\\}\\|\\{(.*?)\\}\\| [ ]*([0-9]*)[ ]*\\}\\}$");

    public static void main(String[] args) {
        BufferedReader lector;
        ArrayList<Articulo> Articulos=new ArrayList<Articulo>();
        HashMap<String,String> DatosPedido=new HashMap<String,String>();

        /*
        * 1er paso: cargamos el archivo para poder procesarlo línea a línea
        * para ello nos apoyamos en la clase BufferedReader, que con el método
        * readLine nos permite recorrer todo el archivo línea a línea.
        */
    }
}
```

```

    if (args.length > 0) {
        lector = cargarArchivo(args[0]);
    } else {
        lector = cargarArchivo();
    }

    if (lector == null) {
        /* Si no se ha podido cargar el archivo, no continúa con el
        * procesado, simplemente termina la ejecución. */
        System.out.println("No se ha podido cargar el archivo.");
    } else {
        /* Si ha podido cargar el archivo, continúa el procesado de línea
        * a línea. */
        String linea;
        try {
            linea = lector.readLine();
            while (linea != null) {
                procesarLinea(linea, DatosPedido, Articulos);
                linea = lector.readLine();
            }
        } catch (IOException ex) {
            System.out.println("Error de entrada y salida.");
        }

        // Mostramos los datos del pedido para ver si son correctos.
        for (String etiqueta: DatosPedido.keySet())
        {
            System.out.println("Dato pedido-->" + etiqueta + ":" + DatosPedido.get(etiqueta));
        }

        // Mostramos los datos de los articulos para ver si son correctos.
        for (Articulo ar: Articulos)
        {
            System.out.print("articulo codigo='" + ar.codArticulo + "' ");
            System.out.print("descripcion='" + ar.descripcion + "' ");
            System.out.println("cantidad='" + ar.cantidad + "'");
        }
    }
}

/**
 * Procesa una línea del archivo de pedido para detectar que es y
 * extraer la información que contiene.
 * @param linea
 * @param datosPedido Mapa en el que irá metiendo la información del pedido.
 * La llave del mapa será el nombre del campo.
 * @param articulos Lista en la que se irán metiendo los artículos del pedido.
 * @return true si la línea contiene información que corresponde al formato
 * esperado, false en caso contrario.
 */
static boolean procesarLinea(String linea, Map<String, String> datosPedido,
    List<Articulo> articulos) {

    Matcher deteccionSeccion = seccion.matcher(linea);
    Matcher deteccionCampo = campo.matcher(linea);
    Matcher deteccionArticulo = articulo.matcher(linea);
    /* Si el patrón coincide con el de un indicador de comienzo del pedido
    * o de la sección con el listado de artículos, se ejecutará este trozo
    * de código, pues habrá encontrado el patrón. No hace nada,
    * simplemente lo detecta para así no informar de algo raro.
    */
    if (deteccionSeccion.matches()) {
        return true;
    }
    /* Si el patrón coincide con el de un campo con datos del pedido
    entonces meterá tanto el campo como el valor en el mapa.*/
    else if (deteccionCampo.matches()) {
        datosPedido.put(deteccionCampo.group(1).trim().toLowerCase(),
            deteccionCampo.group(2).trim());
        return true;
    }
    /* Si el patrón coincide con el de un artículo, entonces
    guardará los datos del pedido en una clase articulo y lo meterá
    en la lista de artículos.*/
    else if (deteccionArticulo.matches())
    {

```



```

        Artículo n=new Artículo();
        n.codArticulo=deteccionArticulo.group(1).trim();
        n.descripcion=deteccionArticulo.group(2).trim();
        n.cantidad=Integer.parseInt(deteccionArticulo.group(3));
        articulos.add(n);
        return true;
    }
    else { System.out.println("¡Cuidado! Línea no procesable: "+linea); return false; }
}

/**
 * cargarArchivo creará una instancia de la clase BufferedReader que
 * permitirá leer línea a línea el archivo de texto. Si no se ha podido
 * cargar el archivo retornará null.
 * @param name Nombre del archivo a cargar. si el nombre del archivo no
 * se ha pasado por parámetro (valor null) se pedirá al usuario que lo
 * introduzca.
 * @return null si no ha podido cargar el archivo, o la instancia de la
 * clase BufferedReader si dicho archivo se ha podido cargar.
 */
static BufferedReader cargarArchivo(String name) {
    String nombreArchivo = name;
    BufferedReader reader = null;
    if (name == null) {
        System.out.print("Introduce el nombre del archivo:");
        nombreArchivo = entrada.nextLine();
    }
    try {
        FileReader f =new FileReader(nombreArchivo);
        reader = new BufferedReader(f);
    } catch (FileNotFoundException ex) {
        Logger.getLogger(ProcesarArchivo2.class.getName()).log(Level.SEVERE, null, ex);
    }
    return reader;
}

/**
 * Igual que el método BufferedReader cargarArchivo(String name), pero
 * que siempre le pedirá al usuario que lo introduzca.
 * @return null si no ha podido cargar el archivo, y una instancia de BufferedReader
 * en otro caso.
 */
static BufferedReader cargarArchivo() {
    return cargarArchivo(null);
}
}

```

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.

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

Los algoritmos de ordenación ordenan los elementos en orden natural, siempre que Java sepa como ordenarlos.

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

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

Ejemplo de ordenación de un array de números Ejemplo de ordenación de una lista con números.

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

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

11.1.- Algoritmos. (II)

En Java hay dos mecanismos para cambiar la forma en la que los elementos se ordenan. ¿Recuerdas la tarea que 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 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 por 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 Artículo 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 `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 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);`”

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

11.2.- Algoritmos. (III)

¿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:

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>
Busqueda 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, extremadamente útil. No se especifica el tipo de lista retornado (no es ArrayList ni LinkedList), solo se especifica que retorna una lista que implementa la interfaz <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<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 Collections, sino propio de la interfaz Collection. Es conveniente que sepas de su existencia.	Para este ejemplo, supondremos que los elementos de la lista son números, dado que hay que crear 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 dado que es necesario conocer el funcionamiento de los arrays y de las expresiones regulares para su uso, no se ha podido

ver hasta ahora. 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:

```
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!

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

[http://www.youtube.com/watch?v= B2mEOA7hSo](http://www.youtube.com/watch?v=B2mEOA7hSo)

Vídeo en el que se explica que la búsqueda binaria es una técnica que permite buscar rápidamente números en un array ordenado (aunque también es aplicable a listas ordenadas). En el proceso de búsqueda binaria consiste en dividir en hallar el punto medio del array. Si el valor buscado coincide con el punto medio, entonces se ha encontrado. Si el valor buscado es menor al valor que hay en la mitad del array, entonces el valor buscado estará en la primera mitad del array, se procede a buscar el valor en dicha mitad. Si el valor buscado es mayor al valor que hay en la mitad del array, entonces el valor buscado estará en la segunda mitad del array, y se procede a buscar el valor en dicha mitad.

Para encontrar el valor en cada una de las mitades, se procede de igual forma, se busca el punto central y se vuelve a dividir en dos. El proceso continúa hasta que no se pueda volver a realizar una nueva división.

12.- Tratamiento de documentos estructurados XML.

Caso práctico.

Ana ya ha terminado lo principal, ya es capaz de procesar el pedido y de almacenar la información en estructuras de memoria que luego podrá proyectar a un documento XML, incluso ha ordenado los artículos en base al código de artículo, definitivamente era bastante más fácil de lo que ella pensaba. Ahora le toca la tarea más ardua de todas, o al menos así lo ve ella, generar el documento XML con la información del pedido, ¿le resultará muy difícil?

¿Qué es XML? XML es un mecanismo extraordinariamente sencillo para estructurar, almacenar e intercambiar información entre sistemas informáticos.

XML define un lenguaje de etiquetas, muy fácil de entender pero con unas reglas muy estrictas, que permite encapsular información de cualquier tipo para posteriormente ser manipulada. Se ha extendido tanto que hoy día es un estándar en el intercambio de información entre sistemas.

La información en XML va escrita en texto legible por el ser humano, pero no está pensada para que sea leída por un ser humano, sino por una máquina. La información va codificada generalmente en unicode, pero estructurada de forma que una máquina es capaz de procesarla eficazmente. Esto tiene una clara ventaja: si necesitamos modificar algún dato de un documento en XML, podemos hacerlo con un editor de texto plano. Veamos los elementos básicos del XML:

Elemento	Descripción	Ejemplo
Cabecera o declaración del XML.	Es lo primero que encontramos en el documento XML y define cosas como, por ejemplo, la codificación del documento XML (que suele ser ISO-8859-1 o UTF-8) y la versión del estándar XML que sigue nuestro documento XML.	<pre><?xml version="1.0" encoding="ISO-8859-1"?></pre>
Etiquetas.	Una etiqueta es un delimitador de datos, y a su vez, un elemento organizativo. La información va entre las etiquetas de apertura (" <pedido> ") y cierre (" </pedido> "). Fíjate en el nombre de la etiqueta ("pedido"), debe ser el mismo tanto en el cierre como en la apertura, respetando mayúsculas.	<pre><pedido> información del pedido </pedido></pre>
Atributos.	Una etiqueta puede tener asociado uno o más atributos. Siempre deben ir detrás del nombre de la etiqueta, en la etiqueta de apertura, poniendo el nombre del atributo seguido de igual y el valor encerrado entre comillas . Siempre debes dejar al menos un espacio entre los atributos.	<pre><articulo cantidad="20"> información </articulo></pre>
Texto.	Entre el cierre y la apertura de una etiqueta puede haber texto.	<pre><cliente> Muebles Bonitos S.A. </cliente></pre>
Etiquetas sin contenido.	Cuando una etiqueta no tiene contenido, no tiene por qué haber una etiqueta de cierre, pero no debes olvidar poner la barra de cierre (" / ") al final de la etiqueta para indicar que no tiene contenido.	<pre><fecha entrega="1/1/2012" /></pre>
Comentario.	Es posible introducir comentarios en XML y estos van dirigidos generalmente a un ser humano que lee directamente el documento XML.	<pre><!-- comentario --></pre>

El nombre de la etiqueta y de los nombres de los atributos no deben tener espacios. También es conveniente evitar los puntos, comas y demás caracteres de puntuación. En su lugar se puede usar el guión bajo ("**<pedido enviado> ... </pedido enviado>**").

Señala las líneas que no serían elementos XML válidos.

```

☒ <cliente>Informática Elegante </cliente>
☐ <!-- -->
☒ <pedido fechaentrega=25 />
☐ <direccion_entrega>sin especificar</direccion_entrega>

```

12.1.- ¿Qué es un documento XML?

Los documentos XML son documentos que **solo** utilizan los elementos expuestos en el apartado anterior (declaración, etiquetas, comentarios, etc.) de **forma estructurada**. Siguen una estructura de árbol, pseudo-jerárquica, permitiendo agrupar la información en diferentes niveles, que van desde la raíz a las hojas.

Para comprender la estructura de un documento XML vamos a utilizar una terminología afín a la forma en la cual procesaremos los documentos XML. Un documento XML está compuesto desde el punto de vista de programación por nodos, por nodos que pueden (o no) contener otros nodos. Todo es un nodo:

- ✓ El par formado por la etiqueta de apertura ("`<etiqueta>`") y por la de cierre ("`</etiqueta>`"), junto con todo su contenido (elementos, atributos y texto de su interior) es un nodo llamado elemento (`Element` desde el punto de vista de programación). Un elemento puede contener otros elementos, es decir, puede contener en su interior subetiquetas, de forma anidada.
- ✓ Un atributo es un nodo especial llamado atributo (`Attr` desde el punto de vista de programación), que solo puede estar dentro de un elemento (concretamente dentro de la etiqueta de apertura).
- ✓ El texto es un nodo especial llamado texto (`Text`), que solo puede estar dentro de una etiqueta.
- ✓ Un comentario es un nodo especial llamado comentario (`Comment`), que puede estar en cualquier lugar del documento XML.
- ✓ Y por último, un documento es un nodo que contiene una jerarquía de nodos en su interior. Está formado opcionalmente por una declaración, opcionalmente por uno o varios comentarios y **obligatoriamente por un único elemento**.

Esto es un poco lioso, ¿verdad? Vamos a clarificarlo con ejemplos. Primero, tenemos que entender la diferencia entre nodos padre y nodos hijo. Un elemento (par de etiquetas) puede contener varios nodos hijo, que pueden ser texto u otros elementos. Por ejemplo:

```

<padre att1="valor" att2="valor">
  texto 1
  <ethija> texto 2 </ethija>
</padre>

```

En el ejemplo anterior, el elemento padre tendría dos hijos: el texto "*texto 1*", sería el primer hijo, y el elemento etiquetado como "*ethija*", el segundo. Tendría también dos atributos, que sería nodos hijo también pero que se consideran especiales y la forma de acceso es diferente. A su vez, el elemento "*ethija*" tiene un nodo hijo, que será el texto "*texto 2*". ¿Fácil no?

Ahora veamos el conjunto, un documento estará formado, como se dijo antes, por algunos elementos opcionales, y obligatoriamente por un único elemento (es decir, por único par de etiquetas que lo engloba todo) que contendrá internamente el resto de información como nodos hijo. Por ejemplo:

```

<?xml version="1.0" encoding="ISO-8859-1"?>
<pedido>
  <cliente> texto </cliente>
  <codCliente> texto </codCliente>
  ...
</pedido>

```

La etiqueta pedido del ejemplo anterior, será por tanto el elemento raíz del documento y dentro de él estará toda la información del documento XML. Ahora seguro que es más fácil, ¿no?

En el siguiente enlace podrás encontrar una breve guía de las tecnologías asociadas a XML actuales. Son bastantes y su potencial es increíble.

<http://www.w3c.es/divulgacion/guiasbreves/tecnologiasxml>

Tecnologías XML	
XML Essentials	XML está compuesto por un conjunto de tecnologías esenciales como el conjunto de información y espacios de nombre. Abordan problemas cuando se utilizan XML en contextos de aplicaciones específicas.
Schema	Las descripciones formales de los vocabularios crean flexibilidad en entornos de creación y las cadenas de control de calidad. XML Schema del W3C, SML, y las tecnologías de enlace de datos proporcionan las herramientas para el control de calidad de los datos XML.
Security	La manipulación de datos con XML requiere a veces de integridad, autenticación y privacidad. Firma XML, cifrado y XKMS puede ayudar a crear un entorno seguro para XML.
Transformation	Muy frecuentemente uno quiere transformar el contenido XML en otro formato (incluyendo otros formatos XML). XSLT y XPath son herramientas poderosas para la creación de diferentes representaciones de contenido XML.
Query	XQuery (soportado por XPath) es un lenguaje de consulta para extraer datos del XML, similar a las reglas de SQL para las bases de datos, o SPARQL para la semántica Web.
Components	The XML ecosystem is using additional tools to create a richer environment for using and manipulating XML documents. These components include style sheets, xlink xml:id, xinclude, xpointer, xforms, xml fragments, and events.
Processing	Un modelo de procesamiento define qué operaciones deben llevarse a cabo en qué orden de un documento XML.
Internationalization	W3C ha trabajado con la comunidad en la internacionalización de XML, por ejemplo, para especificar el idioma del contenido XML.
Publishing	XML surgió de la comunidad de publicación técnica. El uso de XSL-FO para publicar incluso grandes o complejos documentos XML a HTML multilingües, los formatos PDF o de otro tipo; incluyen diagramas y fórmulas MathML SVG en la salida.

12.2.- Librerías para procesar documentos XML. (I)

¿Quién establece las bases del XML? Pues el W3C o *World Wide Web Consortium* es la entidad que establece las bases del XML. Dicha entidad, además de describir como es el XML internamente, define un montón de tecnologías estándar adicionales para verificar, convertir y manipular documentos XML. Nosotros no vamos a explorar todas las tecnologías de XML aquí (son muchísimas), solamente vamos a usar dos de ellas, aquellas que nos van a permitir manejar de forma simple un documento XML:

- ✓ **Procesadores de XML.** Son librerías para leer documentos XML y comprobar que están bien formados. En Java, el procesador más utilizado es SAX, lo usaremos pero sin percatarnos casi de ello.
- ✓ **XML DOM.** Permite transformar un documento XML en un modelo de objetos (de hecho DOM significa *Document Object Model*), accesible cómodamente desde el lenguaje de programación. DOM almacena cada elemento, atributo, texto, comentario, etc. del documento XML en una estructura tipo árbol compuesta por nodos fácilmente accesibles, sin perder la jerarquía del documento. A partir de ahora, la estructura DOM que almacena un XML la llamaremos árbol o jerarquía de objetos DOM.

En Java, estas y otras funciones están implementadas en la librería JAXP (*Java API for XML Processing*), y ya van incorporadas en la edición estándar de Java (Java SE). En primer lugar vamos a

ver como convertir un documento XML a un árbol DOM, y viceversa, para después ver cómo manipular desde Java un árbol DOM.

Para cargar un documento XML tenemos que hacer uso de un procesador de documentos XML (conocidos generalmente como *parsers*) y de un constructor de documentos DOM. Las clases de Java que tendremos que utilizar son:

- ✓ `javax.xml.parsers.DocumentBuilder`: será el procesador y transformará un documento XML a DOM, se le conoce como constructor de documentos.
- ✓ `javax.xml.parsers.DocumentBuilderFactory`: permite crear un constructor de documentos, es una fábrica de constructores de documentos.
- ✓ `org.w3c.dom.Document`: una instancia de esta clase es un documento XML pero almacenado en memoria siguiendo el modelo DOM. Cuando el parser procesa un documento XML creará una instancia de esta clase con el contenido del documento XML.

Ahora bien, ¿esto cómo se usa? Pues muy fácil, en pocas líneas (no olvides importar las librerías):

```
try {
    // 1º Creamos una nueva instancia de una fabrica de constructores de documentos.
    DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
    // 2º A partir de la instancia anterior, fabricamos un constructor de documentos, que
    // procesará el XML.
    DocumentBuilder db = dbf.newDocumentBuilder();
    // 3º Procesamos el documento (almacenado en un archivo) y lo convertimos en un árbol DOM.
    Document doc=db.parse(CaminoAArchivoXml);
} catch (Exception ex) {
    System.out.println(";Error! No se ha podido cargar el documento XML.");
}
```

Es un poco enrevesado pero no tiene mucha complicación, es un pelín más complicado para hacer el camino inverso (pasar el DOM a XML). Este proceso puede generar hasta tres tipos de excepciones diferentes. La más común, que el documento XML esté mal formado, por lo que tienes que tener cuidado con la sintaxis XML.

¿Cuál es la función de la clase `org.w3c.dom.Document`?



Procesar el documento XML



Almacenar el documento XML en un modelo de objetos accesible desde Java



Fabricar un nuevo constructor de documentos

12.2.1.- Librerías para procesar documentos XML. (II)

¿Y cómo paso la jerarquía o árbol de objetos DOM a XML? En Java esto es un pelín más complicado que la operación inversa, y requiere el uso de un montón de clases del paquete `java.xml.transform`, pues la idea es transformar el árbol DOM en un archivo de texto que contiene el documento XML.

Las clases que tendremos que usar son:

- ✓ `javax.xml.transform.TransformerFactory`. Fábrica de transformadores, permite crear un nuevo transformador que convertirá el árbol DOM a XML.
- ✓ `javax.xml.transform.Transformer`. Transformador que permite pasar un árbol DOM a XML.
- ✓ `javax.xml.transform.TransformerException`. Excepción lanzada cuando se produce un fallo en la transformación.
- ✓ `javax.xml.transform.OutputKeys`. Clase que contiene opciones de salida para el transformador. Se suele usar para indicar la codificación de salida (generalmente UTF-8) del documento XML generado.
- ✓ `javax.xml.transform.dom.DOMSource`. Clase que actuará de intermediaria entre el árbol DOM y el transformador, permitiendo al transformador acceder a la información del árbol DOM.
- ✓ `javax.xml.transform.stream.StreamResult`. Clase que actuará de intermediaria entre el transformador y el archivo o String donde se almacenará el documento XML generado.

- ✓ `java.io.File`. Clase que, como posiblemente sabrás, permite leer y escribir en un archivo almacenado en disco. El archivo será obviamente el documento XML que vamos a escribir en el disco.

Esto es un poco lioso, ¿o no? No lo es tanto cuando se ve un ejemplo de cómo realizar el proceso de transformación de árbol DOM a XML, veamos el ejemplo:

```
try {
    // 1º Creamos una instancia de la clase File para acceder al archivo donde guardaremos el XML.
    File f=new File(CaminoAlArchivoXML);
    //2º Creamos una nueva instancia del transformador a través de la fábrica de transformadores.
    Transformer transformer = TransformerFactory.newInstance().newTransformer();
    //3º Establecemos algunas opciones de salida, como por ejemplo, la codificación de salida.
    transformer.setOutputProperty(OutputKeys.INDENT, "yes");
    transformer.setOutputProperty(OutputKeys.ENCODING, "UTF-8");
    //4º Creamos el StreamResult, intermediaria entre el transformador y el archivo de destino.
    StreamResult result = new StreamResult(f);
    //5º Creamos el DOMSource, intermediaria entre el transformador y el árbol DOM.
    DOMSource source = new DOMSource(doc);
    //6º Realizamos la transformación.
    transformer.transform(source, result);
} catch (TransformerException ex) {
    System.out.println("Error! No se ha podido llevar a cabo la transformación.");
}
```

A continuación te adjuntamos, cortesía de la casa, una de esas clases anti-estrés... utilízala todo lo que quieras, con ella podrás convertir un documento XML a árbol DOM y viceversa de forma sencilla. El documento XML puede estar almacenado en un archivo o en una cadena de texto (e incluso en Internet, para lo que necesitas el URI). Los métodos estáticos DOM2XML te permitirán pasar el árbol DOM a XML, y los métodos String2DOM y XML2DOM te permitirán pasar un documento XML a un árbol DOM.

También contiene el método crearDOMVacio que permite crear un árbol DOM vacío, útil para empezar un documento XML de cero.

```
package javaapplication1;

import java.io.ByteArrayInputStream;
import java.io.File;
import java.io.StringWriter;
import java.util.logging.Level;
import java.util.logging.Logger;
import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.parsers.ParserConfigurationException;
import javax.xml.transform.OutputKeys;
import javax.xml.transform.Transformer;
import javax.xml.transform.TransformerException;
import javax.xml.transform.TransformerFactory;
import javax.xml.transform.dom.DOMSource;
import javax.xml.transform.stream.StreamResult;
import org.w3c.dom.Document;

/**
 * Utilidades para pasar árboles DOM a documentos XML y viceversa.
 * @author Salvador Romero Villegas
 */
public class DOMUtil {

    /**
     * Carga un archivo con un documento XML a un árbol DOM.
     * @param CaminoAArchivoXml puede ser un archivo local de tu disco duro
     * o una URI de Internet (http://...).
     * @return el documento DOM o null si no se ha podido cargar el documento.
     */
    public static Document XML2DOM (String CaminoAArchivoXml)
    {
        Document doc=null;
        try {
            DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
            DocumentBuilder db = dbf.newDocumentBuilder();
            doc=db.parse(CaminoAArchivoXml);
        } catch (Exception ex) {
```

```

        Logger.getLogger(DOMUtil.class.getName()).log(Level.SEVERE, null, ex);
    }
    return doc;
}

/**
 * Convierte una cadena que contiene un documento XML a un árbol DOM.
 * @param documentoXML cadena que contiene el documento XML.
 * @return El árbol DOM o null si no se ha podido convertir.
 */
public static Document String2DOM (String documentoXML)
{
    ByteArrayInputStream bais=new ByteArrayInputStream(documentoXML.getBytes());
    Document doc=null;
    try {
        DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
        DocumentBuilder db = dbf.newDocumentBuilder();
        doc=db.parse(bais);

    } catch (Exception ex) {
        Logger.getLogger(DOMUtil.class.getName()).log(Level.SEVERE, null, ex);
    }
    return doc;
}

/**
 * Convierte un árbol DOM a una cadena que contiene un documento XML.
 * @param doc Árbol DOM.
 * @return null si no se ha podido convertir o la cadena con el documento
 * en XML si se ha podido convertir.
 */
public static String DOM2XML (Document doc)
{
    String xmlString=null;
    try {
        Transformer transformer = TransformerFactory.newInstance().newTransformer();
        transformer.setOutputProperty(OutputKeys.INDENT, "yes");
        transformer.setOutputProperty(OutputKeys.ENCODING, "UTF-8");
        StreamResult result = new StreamResult(new StringWriter());
        DOMSource source = new DOMSource(doc);
        transformer.transform(source, result);
        xmlString = result.getWriter().toString();
    } catch (TransformerException ex) {
        Logger.getLogger(DOMUtil.class.getName()).log(Level.SEVERE, null, ex);
        xmlString=null;
    }
    return xmlString;
}

/**
 * Convierte un árbol DOM a XML y lo guarda en un archivo.
 * @param doc Documento a convertir en XML.
 * @param CaminoAlArchivoXML Camino o path para llegar al archivo en el
 * disco.
 * @return true si se ha podido convertir y false en cualquier otra situación.
 */
public static boolean DOM2XML (Document doc, String CaminoAlArchivoXML)
{
    try {
        File f=new File(CaminoAlArchivoXML);
        Transformer transformer = TransformerFactory.newInstance().newTransformer();
        transformer.setOutputProperty(OutputKeys.INDENT, "yes");
        transformer.setOutputProperty(OutputKeys.ENCODING, "UTF-8");
        StreamResult result = new StreamResult(f);
        DOMSource source = new DOMSource(doc);
        transformer.transform(source, result);
        return true;
    } catch (TransformerException ex) {
        Logger.getLogger(DOMUtil.class.getName()).log(Level.SEVERE, null, ex);
    }
    return false;
}

/**
 * Crea un árbol DOM vacío.
 * @param etiquetaRaiz Nombre de la etiqueta raíz del árbol DOM, donde
 * estará contenida el resto del documento.

```

```

    * @return Retornará el documento creado o null si se ha producido algún
    * error.
    */
    public static Document crearDOMVacio(String etiquetaRaiz)
    {
        DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
        DocumentBuilder db;
        try {
            db = dbf.newDocumentBuilder();
            Document d=db.newDocument();
            d.appendChild(d.createElement(etiquetaRaiz));
            return d;
        } catch (ParserConfigurationException ex) {
            Logger.getLogger(DOMUtil.class.getName()).log(Level.SEVERE, null, ex);
        }
        return null;
    }
}

```

12.3.-Manipulación de documentos XML.(I)

Bien, ahora sabes cargar un documento XML a DOM y de DOM a XML, pero, ¿cómo se modifica el árbol DOM? Como ya se dijo antes, un árbol DOM es una estructura en árbol, jerárquica como cabe esperar, formada por nodos de diferentes tipos. El funcionamiento del modelo de objetos DOM es establecido por el organismo W3C, lo cual tiene una gran ventaja, el modelo es prácticamente el mismo en todos los lenguajes de programación.

En Java, prácticamente todas las clases que vas a necesitar para manipular un árbol DOM están en el paquete `org.w3c.dom`. Si vas a hacer un uso muy intenso de DOM es conveniente que hagas una importación de todas las clases de este paquete (`"import org.w3c.dom.*;"`).

Tras convertir un documento XML a DOM lo que obtenemos es una instancia de la clase `org.w3c.dom.Document`. Esta instancia será el nodo principal que contendrá en su interior toda la jerarquía del documento XML. Dentro de un documento o árbol DOM podremos encontrar los siguientes tipos de clases:

- ✓ `org.w3c.dom.Node (Nodo)`. Todos los objetos contenidos en el árbol DOM son nodos. La clase `Document` es también un tipo de nodo, considerado el nodo principal.
- ✓ `org.w3c.dom.Element (Elemento)`. Corresponde con cualquier par de etiquetas (`"<pedido></pedido>"`) y todo su contenido (atributos, texto, subetiquetas, etc.).
- ✓ `org.w3c.dom.Attr (Atributo)`. Corresponde con cualquier atributo.
- ✓ `org.w3c.dom.Comment (Comentario)`. Corresponde con un comentario.
- ✓ `org.w3c.dom.Text (Texto)`. Corresponde con el texto que encontramos dentro de dos etiquetas.

¿A qué te eran familiares? Claro que sí. Estas clases tendrán diferentes métodos para acceder y manipular la información del árbol DOM. A continuación vamos a ver las operaciones más importantes sobre un árbol DOM.

En todos los ejemplos, *"doc"* corresponde con una instancia de la clase `Document`.

Obtener el elemento raíz del documento.

Como ya sabes, los documentos XML deben tener obligatoriamente un único elemento (`"<pedido></pedido>"` por ejemplo), considerado el elemento raíz, dentro del cual está el resto de la información estructurada de forma jerárquica. Para obtener dicho elemento y poder manipularlo podemos usar el método `getDocumentElement`.

```
Element raiz=doc.getDocumentElement();
```

Buscar un elemento en toda la jerarquía del documento.

Para realizar esta operación se puede usar el método `getElementsByTagName` disponible tanto en la clase `Document` como en la clase `Element`. Dicha operación busca un elemento por el nombre de la etiqueta y retorna una lista de nodos (`NodeList`) que cumplen con la condición. Si se usa en la clase `Element`, solo buscará entre las subetiquetas (subelementos) de dicha clase (no en todo el documento).

```

NodeList nl=doc.getElementsByTagName("cliente");
Element cliente;
if (nl.getLength()==1) cliente=(Element)n2.item(0);

```

El método `getLength()` de la clase `NodeList`, permite obtener el número de elementos (longitud de la lista) encontrados cuyo nombre de etiqueta es coincidente. El método `item` permite acceder a cada uno de los elementos encontrados, y se le pasa por argumento el índice del elemento a obtener (empezando por cero y acabando por longitud menos uno). Fíjate que es necesario hacer una conversión de tipos después de invocar el método `item`. Esto es porque la clase `NodeList` almacena un listado de nodos (`Node`), sin diferenciar el tipo.

Dado el siguiente código XML: “<pedido><!--Datos del pedido--></pedido>” indica que elementos DOM encontramos en el mismo.

- ☒ `org.w3c.dom.Element`
- ☒ `org.w3c.dom.Node`
- ☐ `org.w3c.dom.Text`
- ☒ `org.w3c.dom.Comment`

12.3.1.-Manipulación de documentos XML.(II)

¿Y qué más operaciones puedo realizar sobre un árbol DOM? Veámoslas.

Obtener la lista de hijos de un elemento y procesarla.

Se trata de obtener una lista con los nodos hijo de un elemento cualquiera, estos pueden ser un sub-elemento (sub-etiqueta) o texto. Para sacar la lista de nodos hijo se puede usar el método

`getChildNodes()`:

```
NodeList nl=doc.getDocumentElement().getChildNodes();
for (int i=0; i<nl.getLength();i++) {
    Node n=nl.item(i);
    switch (n.getNodeType()){
        case Node.ELEMENT_NODE: Element e=(Element)n;
            System.out.println("Etiqueta:" + e.getTagName());
            break;
        case Node.TEXT_NODE: Text t=(Text)n;
            System.out.println("Texto:" + t.getWholeText());
            break;
    }
}
```

En el ejemplo anterior se usan varios métodos. El método “`getNodeTypes()`” de la clase `Node` permite saber de qué tipo de nodo se trata, generalmente texto (`Node.TEXT_NODE`) o un sub-elemento (`Node.ELEMENT_NODE`). De esta forma podremos hacer la conversión de tipos adecuada y gestionar cada elemento según corresponda. También se usa el método “`getTagName`” aplicado a un elemento, lo cual permitirá obtener el nombre de la etiqueta, y el método “`getWholeText`” aplicado a un nodo de tipo texto (`Text`), que permite obtener el texto contenido en el nodo.

Añadir un nuevo elemento hijo a otro elemento.

Hemos visto como mirar que hay dentro de un documento XML pero no hemos visto como añadir cosas a dicho documento. Para añadir un sub-elemento o un texto a un árbol DOM, primero hay que crear los nodos correspondientes y después insertarlos en la posición que queramos. Para crear un nuevo par de etiquetas o elemento (`Element`) y un nuevo nodo texto (`Text`), lo podemos hacer de la siguiente forma:

```
Element dirTag=doc.createElement("Direccion_entrega")
Text dirTxt=doc.createTextNode("C/Perdida S/N");
```

Ahora los hemos creado, pero todavía no los hemos insertado en el documento. Para ello podemos hacerlo usando el método `appendChild` que añadirá el nodo (sea del tipo que sea) al final de la lista de hijos del elemento correspondiente:

```
dirTag.appendChild(dirTxt);
doc.getDocumentElement().appendChild(dirTag);
```

En el ejemplo anterior, el texto se añade como hijo de la etiqueta “Direccion_entrega”, y a su vez, la etiqueta “Direccion_entrega” se añade como hijo, al final del todo, de la etiqueta o elemento raíz del documento. Aparte del método `appendChild`, que siempre insertará al final, puedes utilizar los siguientes métodos para insertar nodos dentro de un árbol DOM (todos se usan sobre la clase `Element`):

- ✓ `insertBefore (Node nuevo, Node referencia)`. Insertará un nodo nuevo antes del nodo de referencia.
- ✓ `replaceChild (Node nuevo, Node anterior)`. Sustituye un nodo (anterior) por uno nuevo.

En el siguiente enlace encontrarás a la documentación del API de DOM para Java, con todas las funciones de cada clase.

<http://docs.oracle.com/javase/1.4.2/docs/api/org/w3c/dom/package-summary.html>

12.3.2.-Manipulación de documentos XML.(III)

Seguimos con las operaciones sobre árboles DOM. ¿Sabrías cómo eliminar nodos de un árbol? ¿No? Vamos a descubrirlo.

Eliminar un elemento hijo de otro elemento.

Para eliminar un nodo, hay que recurrir al nodo padre de dicho nodo. En el nodo padre se invoca el método `removeChild`, al que se le pasa la instancia de la clase `Element` con el nodo a eliminar (no el nombre de la etiqueta, sino la instancia), lo cual implica que primero hay que buscar el nodo a eliminar, y después eliminarlo. Veamos un ejemplo:

```
NodeList nl3=doc.getElementsByTagName("Direccion entrega");
for (int i=0;i<nl3.getLength();i++){
    Element e=(Element)nl3.item(i);
    Element parent=(Element)e.getParentNode();
    parent.removeChild(e);
}
```

En el ejemplo anterior se eliminan todas las etiquetas, estén donde estén, que se llamen “Direccion_entrega”. Para ello ha sido necesario buscar todos los elementos cuya etiqueta sea esa (como se explico en ejemplos anteriores), recorrer los resultados obtenidos de la búsqueda, obtener el nodo padre del hijo a través del método `getParentNode`, para así poder eliminar el nodo correspondiente con el método `removeChild`.

No es obligatorio obviamente invocar al método “`getParentNode`” si el nodo padre es conocido. Por ejemplo, si el nodo es un hijo del elemento o etiqueta raíz, hubiera bastado con poner lo siguiente:

```
doc.getDocumentElement().removeChild(e);
```

Cambiar el contenido de un elemento cuando solo es texto.

Los métodos `getTextContent` y `setTextContent`, aplicado a un elemento, permiten respectivamente acceder al texto contenido dentro de un elemento o etiqueta. Tienes que tener cuidado, porque utilizar “`setTextContent`” significa eliminar cualquier hijo (sub-elemento por ejemplo) que previamente tuviera la etiqueta. Ejemplo:

```
Element nuevo=doc.createElement("direccion_recogida").setTextContent("C/Del Medio S/N");
System.out.println(nuevo.getTextContent());
```

De los siguientes tipos de nodos, marca aquellos en los que no se pueda ejecutar dos veces el método “`appendChild`”:

- ☒ Document
- ☐ Element
- ☒ Text
- ☒ Attr

12.3.3.-Manipulación de documentos XML.(IV)

Caso práctico.

El documento XML que tiene que generar Ana tiene que seguir un formato específico, para que la otra aplicación sea capaz de entenderlo. Esto significa que los nombres de las etiquetas tienen que ser unos concretos para cada dato del pedido. Esta ha sido quizás la parte que más complicada, ver como encajar cada dato del mapa con su correspondiente etiqueta en XML, pero ha conseguido resolverlo de forma elegante. De hecho, ya ha terminado su tarea, ¿quieres ver el resultado?

```
import java.io.BufferedReader;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;
import java.util.ArrayList;
import java.util.Collections;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.Scanner;
import java.util.logging.Level;
import java.util.logging.Logger;
import java.util.regex.Matcher;
import java.util.regex.Pattern;
import org.w3c.dom.*;

/*
 * Clase destinada a almacenar los datos de un artículo.
 */
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);
    }
}

/* Clase que se encarga de procesar un pedido.
 * El archivo con el pedido debe estar en codificación UTF8, sino no funciona
 * bien.
 * Tu sistema debe soportar de forma nativa la coficación UTF8, sino no
 * funcionará del todo bien.
 */
public class ProcesarArchivo3 {

    /* Entrada contendrá una instancia de la clase Scanner que permitirá
    leer las teclas pulsadas desde teclado */
    static Scanner entrada = new Scanner(System.in);
    /* Definimos las expresiones regulares que usaremos una y otra vez para
    cada línea del pedido. La expresión regular "seccion" permite detectar
    si hay un comienzo o fin de pedido, y la expresión campo, permite detectar
    si hay un campo con información del pedido. */
    static Pattern seccion = Pattern.compile("^##[ ]*(FIN)?[ ]*(PEDIDO|ARTICULOS)[ ]*##$");
    static Pattern campo = Pattern.compile("^(.+):.*\\{(.*?)\\}$");
    static Pattern articulo = Pattern.compile ("^\\{(.*?)\\|(.*?)\\|[ ]*([0-9]*)[ ]*\\}$");

    public static void main(String[] args) {
        BufferedReader lector;
        ArrayList<Articulo> Articulos=new ArrayList<Articulo>();
        HashMap<String,String> DatosPedido=new HashMap<String,String>();

        /*
        * 1er paso: cargamos el archivo para poder procesarlo línea a línea
        * para ello nos apoyamos en la clase BufferedReader, que con el método
        * readLine nos permite recorrer todo el archivo línea a línea.
        */
        if (args.length > 0) {
            lector = cargarArchivo(args[0]);
        } else {
            lector = cargarArchivo();
        }
    }
}
```



```

if (lector == null) {
    /* Si no se ha podido cargar el archivo, no continúa con el
    * procesado, simplemente termina la ejecución. */
    System.out.println("No se ha podido cargar el archivo.");
} else {
    /* 2º Paso: si ha podido cargar el archivo, continúa el procesado c
    * de línea a línea. */
    String linea;
    try {
        linea = lector.readLine();
        while (linea != null) {
            procesarLinea(linea,DatosPedido,Articulos);
            linea = lector.readLine();
        }
    } catch (IOException ex) {
        System.out.println("Error de entrada y salida.");
    }

    // 3er paso: Ordenamos los artículos por código.
    Collections.sort(Articulos);

    /* 4er Paso: Pasamos el pedido a árbol DOM. */
    // Creamos un árbol DOM vacío
    Document doc=DOMUtil.crearDOMVacio("pedido");
    // Pasamos los datos del pedido al DOM
    pasarPedidoAXML(doc, DatosPedido, Articulos);
    // Guardamos el XML en un archivo:
    String salida;
    if (args.length>1) salida=args[1];
    else {
        System.out.print("Introduce el archivo de salida: ");
        salida=entrada.nextLine();
    }
    DOMUtil.DOM2XML(doc,salida);
}

}

/**
 * Procesa una línea del archivo de pedido para detectar que es y
 * extraer la información que contiene.
 * @param linea
 * @param datosPedido Mapa en el que irá metiendo la información del pedido.
 * La llave del mapa será el nombre del campo.
 * @param articulos Lista en la que se irán metiendo los artículos del pedido.
 * @return true si la línea contiene información que corresponde al formato
 * esperado, false en caso contrario.
 */
static boolean procesarLinea(String linea, Map<String,String> datosPedido,
    List<Articulo> articulos) {

    Matcher deteccionSeccion = seccion.matcher(linea);
    Matcher deteccionCampo = campo.matcher(linea);
    Matcher deteccionArticulo= articulo.matcher(linea);
    /* Si el patrón coincide con el de un indicador de comienzo del pedido
    * o de la sección con el listado de artículos, se ejecutará este trozo
    * de código, pues habrá encontrado el patrón. No hace nada,
    * simplemente lo detecta para así no informar de algo raro.
    */
    if (deteccionSeccion.matches()) {
        return true;
    }
    /* Si el patrón coincide con el de un campo con datos del pedido
    entonces meterá tanto el campo como el valor en el mapa.*/
    else if (deteccionCampo.matches()) {
        datosPedido.put(deteccionCampo.group(1).trim().toLowerCase(),
            deteccionCampo.group(2).trim());
        return true;
    }
    /* Si el patrón coincide con el de un artículo, entonces
    guardará los datos del pedido en una clase articulo y lo meterá
    en la lista de artículos.*/
    else if (deteccionArticulo.matches())
    {
        Articulo n=new Articulo();
        n.codArticulo=deteccionArticulo.group(1).trim();
    }
}

```

```

        n.descripcion=deteccionArticulo.group(2).trim();
        n.cantidad=Integer.parseInt(deteccionArticulo.group(3));
        articulos.add(n);
        return true;
    }
    else { System.out.println(";Cuidado! Línea no procesable: "+linea); return false; }
}

/**
 * cargarArchivo creará una instancia de la clase BufferedReader que
 * permitirá leer línea a línea el archivo de texto. Si no se ha podido
 * cargar el archivo retornará null.
 * @param name Nombre del archivo a cargar. si el nombre del archivo no
 * se ha pasado por parámetro (valor null) se pedirá al usuario que lo
 * introduzca.
 * @return null si no ha podido cargar el archivo, o la instancia de la
 * clase BufferedReader si dicho archivo se ha podido cargar.
 */
static BufferedReader cargarArchivo(String name) {
    String nombreArchivo = name;
    BufferedReader reader = null;
    if (name == null) {
        System.out.print("Introduce el nombre del archivo con el pedido: ");
        nombreArchivo = entrada.nextLine();
    }
    try {
        FileReader f =new FileReader(nombreArchivo);
        reader = new BufferedReader(f);
    } catch (FileNotFoundException ex) {
        Logger.getLogger(ProcesarArchivo3.class.getName()).log(Level.SEVERE, null, ex);
    }
    return reader;
}

/**
 * Igual que el método BufferedReader cargarArchivo(String name), pero
 * que siempre le pedirá al usuario que lo introduzca.
 * @return null si no ha podido cargar el archivo, y una instancia de BufferedReader
 * en otro caso.
 */
static BufferedReader cargarArchivo() {
    return cargarArchivo(null);
}

/**
 * Función que pasa los datos del pedido, almacenados en un mapa y
 * una lista, a un documento XML.
 * @param doc Documento DOM donde se almacenará toda la información.
 * @param datosPedido Mapa con los datos del pedido.
 * @param articulos Lista con los articulos del pedido.
 */
static void pasarPedidoAXML(Document doc,
    Map<String,String> datosPedido,
    List<Articulo> articulos)
{
    for (String key:datosPedido.keySet())
    {
        //Eliminamos espacios y acentos de cada llave para que no haya problemas de
chequeo
        Element e=null;
        String key2=key.trim().replaceAll("\\s", " ").toLowerCase();
        key2=key2.replace("á", "a");
        key2=key2.replace("é", "e");
        key2=key2.replace("í", "i");
        key2=key2.replace("ó", "o");
        key2=key2.replace("ú", "u");
        if (key2.equals("nombre del contacto"))
            e=doc.createElement("contacto");
        else if (key2.equals("forma de pago"))
            e=doc.createElement("formaPago");
        else if (key2.equals("direccion de factura"))
            e=doc.createElement("dirFacturacion");
        else if (key2.equals("correo electronico del contacto"))
            e=doc.createElement("mail");
        else if (key2.equals("codigo del cliente"))
            e=doc.createElement("codClient");
        else if (key2.equals("cliente"))

```

```

        e=doc.createElement("nombreCliente");
    else if (key2.equals("numero de pedido"))
        e=doc.createElement("numPedido");
    else if (key2.equals("telefono del contacto"))
        e=doc.createElement("telefono");
    else if (key2.equals("fecha preferente de entrega"))
        e=doc.createElement("fechaPrefEntrega");
    else if (key2.equals("direccion de entrega"))
        e=doc.createElement("dirEntrega");

    if (e!=null) {
        e.setTextContent(datosPedido.get(key));
        doc.getDocumentElement().appendChild(e);
    }
    else
    {
        Comment c=doc.createComment("Error procesando "+ key2);
        doc.getDocumentElement().appendChild(c);
    }
}
Element arts=doc.createElement("listaArticulos");
for (Articulo a:articulos)
{
    Element art=doc.createElement("articulo");
    art.setAttribute("codArticulo", a.codArticulo);
    art.setAttribute("cantidad", a.cantidad+"");
    art.setTextContent(a.descripcion);
    arts.appendChild(art);
}
doc.getDocumentElement().appendChild(arts);
}
}

```

Y ya solo queda una cosa, descubrir como manejar los atributos en un árbol DOM.

Atributos de un elemento.

Por último, lo más fácil. Cualquier elemento puede contener uno o varios atributos. Acceder a ellos es sencillísimo, solo necesitamos tres métodos:

`setAttribute` para establecer o crear el valor de un atributo, `getAttribute` para obtener el valor de un atributo y `removeAttribute` para eliminar el valor de un atributo. Veamos un ejemplo:

```

doc.getDocumentElement().setAttribute("urgente","no");
System.out.println(doc.getDocumentElement().getAttribute("urgente"));

```

En el ejemplo anterior se añade el atributo *“urgente”* al elemento raíz del documento con el valor *“no”*, después se consulta para ver su valor. Obviamente se puede realizar en cualquier otro elemento que no sea el raíz. Para eliminar el atributo es tan sencillo como lo siguiente:

```

doc.getDocumentElement().removeAttribute("urgente");

```