UT 6 Aplicación de las estructuras de almacenamiento

Cuando el volumen de datos a manejar por una aplicación es elevado, no basta con utilizar variables.

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

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

1.- Introducción a las estructuras de almacenamiento.

¿Cómo almacenarías en memoria un listado de números del que tienes que extraer el valor máximo? Seguro que te resultaría fácil. Pero, ¿**y si el listado de números no tiene un tamaño fijo, sino que puede variar en tamaño de forma dinámica?** Entonces la cosa se complica.

Las estructuras de almacenamiento, en general, se pueden clasificar de varias formas. Por ejemplo, atendiendo a si pueden almacenar datos de diferente tipo, o si solo pueden almacenar datos de un solo tipo, se pueden distinguir:

- Estructuras con capacidad de **almacenar varios datos del mismo tipo:** varios números, varios caracteres, etc. Ejemplos de estas estructuras son los **arrays,** las **cadenas de caracteres**, las **listas** y los **conjuntos**.
- Estructuras con capacidad de **almacenar varios datos de distinto tipo:** números, fechas, cadenas de caracteres, etc., todo junto dentro de una misma estructura. Ejemplos de este tipo de estructuras son las **clases**.

Otra forma de clasificar las estructuras de almacenamiento va en función de si pueden o no cambiar de tamaño de forma dinámica:

- **Estructuras cuyo tamaño se establece en el momento de la creación o definición** y su tamaño no puede variar después. Ejemplos de estas estructuras son los **arrays** y las matrices (arrays multimensionales).
- **Estructuras cuyo tamaño es variable (conocidas como estructuras dinámicas)**. Su tamaño crece o decrece según las necesidades de forma dinámica. Es el caso de las **listas**, **árboles**, **conjuntos** y, como veremos también, el caso de algunos tipos de cadenas de caracteres.

2.- Cadenas de caracteres.

2.1.1.- Operaciones avanzadas con cadenas de caracteres (II).

Vamos a continuar revisando las operaciones que se pueden realizar con cadenas. Como verás las operaciones a realizar se complican un poco a partir de ahora. En todos los ejemplos la variable **cad** contiene la cadena "**¡Bienvenido!**".

- **int length().** Retorna un número entero que contiene la longitud de una cadena, resultado de contar el número de caracteres por la que esta compuesta. Recuerda que un espacio es también un carácter.
- **char charAt(int pos).** Retorna el caracter ubicado en la posición pasada por parámetro. El caracter obtenido de dicha posición será almacenado en un tipo de dato char. Las posiciones se empiezan a contar desde el 0 (y no desde el 1), y van desde 0 hasta longitud - 1. Por ejemplo, el código siguiente mostraría por pantalla el carácter "v":

```
char t = cad.charAt(5);
System.out.println(t);
```

- **String substring(int beginIndex, int endIndex).** Este método permite extraer una subcadena de otra de mayor tamaño. Una cadena compuesta por todos los caracteres existentes entre la posición beginIndex y la posición endIndex – 1.
- **String substring (int beginIndex).** Cuando al método substring solo le proporcionamos un parámetro, extraerá una cadena que comenzará en el caracter con posición beginIndex e irá hasta el final de la cadena.

```
String subcad = cad.substring(2);
System.out.println(subcad);
```

2.1.2.- Operaciones avanzadas con cadenas de caracteres (III).

¿Cómo comprobarías que la cadena "**3**" es mayor que 0? No puedes comparar directamente una cadena con un número, así que necesitarás aprender cómo convertir cadenas que contienen números a tipos de datos numéricos (**int**, **short**, **long**, **float** o **double**). Esta es una operación habitual en todos los lenguajes de programación, y Java, para este propósito, ofrece los métodos **valueOf**, existentes en todas las clases envoltorio descendientes de la clase **Number: Integer, Long, Short, Float** y **Double**.

Veamos un ejemplo de su uso para un número de doble precisión, para el resto de las clases es similar:

```
String c="1234.5678";

double n;
try {
        n=Double.valueOf(c).doubleValue();
} catch (NumberFormatException e)
{    /* Código a ejecutar si no se puede  convertir */                    }
```

El código escrito esta destinado a transformar la cadena en número, usando el método **valueOf**. Este método lanzará la excepción **NumberFormatException** si no consigue convertir el texto a número.

2.1.3.- Operaciones avanzadas con cadenas de caracteres (IV).

¿Cómo puedo comprobar si dos cadenas son iguales? ¿Qué más operaciones ofrece Java sobre las cadenas?
En la siguiente tabla puedes ver las operaciones más importantes. En todos los ejemplos expuestos, las variables **cad1**, **cad2**y **cad3** son cadenas ya existentes, y la variable **num** es un número entero mayor o igual a cero.

Métodos importantes de la clase String

Método.	Descripción
cad1.equals(cad2)	Cuando se comparan si dos cadenas son iguales, no se debe usar el operador de comparación "==", sino el método equals. Retornará true si son iguales, y false si no lo son.
cad1.trim()	Genera una copia de la cadena eliminando los espacios en blanco anteriores y posteriores de la cadena.
cad1.toLowerCase()	Genera una copia de la cadena con todos los caracteres a minúscula.
cad1.toUpperCase()	Genera una copia de la cadena con todos los caracteres a mayúsculas.

Método.	Descripción
cad1.indexOf(cad2) cad1.indexOf(cad2, num)	Si la cadena o caracter pasado por argumento está contenida en la cadena invocante, retorna su posición, en caso contrario retornará -1. Opcionalmente se le puede indicar la posición a partir de la cual buscar, lo cual es útil para buscar varias apariciones de una cadena dentro de otra.
cad1.contains(cad2)	Retornará true si la cadena pasada por argumento está contenida dentro de la cadena. En caso contrario retornará false.
cad1.startsWith(cad2)	Retornará true si la cadena comienza por la cadena pasada como argumento. En caso contrario retornará false.
cad1.endsWith(cad2)	Retornará true si la cadena acaba por la cadena pasada como argumento. En caso contrario retornará false.
cad1.replace(cad2, cad3)	Generará una copia de la cadena cad1, en la que se reemplazarán todas las apariciones de cad2 por cad3. El reemplazo se hará de izquierda a derecha, por ejemplo: reemplazar "zzz" por "xx" en la cadena "zzzzz" generará "xxzz" y no "zzxx".

2.1.4.- Operaciones avanzadas con cadenas de caracteres (V).

¿En que se diferencian **StringBuilder** de la clase **String**? **Pues básicamente en que la clase StringBuilder permite modificar la cadena que contiene, mientras que la clase String no**. Al realizar operaciones complejas se crea una nueva instancia de la clase **String**.

Veamos un pequeño ejemplo de uso de esta clase. En el ejemplo que vas a ver, se parte de una cadena con errores, que modificaremos para ir haciéndola legible. Lo primero que tenemos que hacer es crear la instancia de esta clase. Se puede inicializar de muchas formas, por ejemplo, partiendo de un literal de cadena:

```
StringBuilder strb=new StringBuilder ("Hoal Muuundo");
```

Y ahora, usando los **métodos append** (insertar al final), **insert**(insertar una cadena o carácter en una posición específica), **delete**(eliminar los caracteres que hay entre dos posiciones) y **replace** (reemplazar los caracteres que hay entre dos posiciones por otros diferentes), rectificaremos la cadena anterior y la haremos correcta:

1. **strb.delete(6,8);** Eliminamos las 'uu' que sobran en la cadena. La primera 'u' que sobra está en la posición 6 (no olvides contar el espacio), y la última 'u' a eliminar está en la posición 7. Para eliminar dichos caracteres de forma correcta hay que pasar como primer argumento la posición 6 (posición inicial) y como segundo argumento la posición 8 (posición contigua al último carácter a eliminar), dado que la posición final no indica el último carácter a eliminar, sino el carácter justo posterior al último que hay que eliminar (igual que ocurría con el método substring).
2. **strb.append ("!");** Añadimos al final de la cadena el símbolo de cierre de exclamación.
3. **strb.insert (0,"¡");** Insertamos en la posición 0, el símbolo de apertura de exclamación.
4. **strb.replace (3,5,"la")**; Reemplazamos los caracteres 'al' situados entre la posición inicial 3 y la posición final 4, por la cadena 'la'. En este método ocurre igual que en los métodos delete y substring, en vez de indicar como posición final la posición 4, se debe indicar justo la posición contigua, es decir 5.

StringBuilder contiene muchos métodos de la clase **String**(**charAt, indexOf, lenght, substring, replace,** etc.), pero no todos, pues son clases diferentes con funcionalidades realmente diferentes.

2.2.- Expresiones regulares (I).

¿Podrías hacer un programa que verificara si un DNI o un NIE es correcto? Seguro que sí. Si te fijas, los números de DNI y los de NIE tienen una estructura fija: X1234567Z (en el caso del NIE) y 1234567Z (en el caso del DNI). Ambos siguen un **patrón** que podría describirse como: una letra inicial opcional (solo presente en los NIE), seguida de una secuencia numérica y finalizando con otra letra. ¿Fácil no?

Pues esta es la función de las expresiones regulares: **permitir comprobar si una cadena sigue o no un patrón preestablecido**. Las expresiones regulares son un mecanismo para describir esos patrones, y se construyen de una forma relativamente sencilla. Existen muchas librerías diferentes para trabajar con expresiones regulares, y casi todas siguen, más o menos, una sintaxis similar, con ligeras variaciones. Dicha sintaxis nos permite indicar el patrón de forma cómoda, como si de una cadena de texto se tratase, en la que determinados símbolos tienen un significado especial. Por ejemplo "**[01]+**" es una expresión regular que permite comprobar si una cadena conforma un número binario. Veamos cuáles son las reglas generales para construir una expresión regular:

- **Podemos indicar que una cadena contiene un conjunto de símbolos fijo**, simplemente poniendo dichos símbolos en el patrón, excepto para algunos símbolos especiales que necesitarán un caracter de escape como veremos más adelante. Por ejemplo, el patrón "**aaa**" admitirá **cadenas** que **contengan tres aes**.
- "[xyz]". Entre **corchetes** podemos indicar **opcionalidad**. Solo uno de los símbolos que hay entre los corchetes podrá aparecer en el lugar donde están los corchetes. Por ejemplo, la expresión regular "aaa[xy]" admitirá como válidas las cadenas "aaax" y la cadena "aaay". **Los corchetes representan una posición de la cadena que puede tomar uno de varios valores posibles.**
- "[a-z]" "[A-Z]" "[a-zA-Z]". Usando el **guión** y los **corchetes** podemos indicar que el patrón **admite cualquier carácter entre la letra inicial y la final.** Es importante que sepas que **se diferencia entre letras mayúsculas y minúsculas**, no son iguales de cara a las expresiones regulares.
- "[0-9]". Y nuevamente, usando un **guión**, podemos indicar que se permite la presencia de un dígito **numérico entre 0 y 9**, cualquiera de ellos, pero solo uno.

Con las reglas anteriores podemos indicar el conjunto de símbolos que admite el patrón y el orden que deben tener. Si una cadena no contiene los símbolos especificados en el patrón, en el mismo orden, entonces la cadena no encajará con el patrón.

Veamos ahora como indicar repeticiones:

- "**a?**". Usaremos el **interrogante** para indicar que un símbolo **puede aparecer una vez o ninguna**. De esta forma la letra "a" podrá aparecer una vez o simplemente no aparecer.
- "**a*".** Usaremos el **asterisco** para indicar que un símbolo **puede aparecer una vez o muchas veces**, pero también ninguna. Cadenas válidas para esta expresión regular serían "aa", "aaa" o "aaaaaaaa".
- "**a+**". Usaremos el símbolo de **suma** para indicar que otro símbolo **debe aparecer al menos una vez,** pudiendo repetirse cuantas veces quiera.
- "**a{1,4}"** . Usando las **llaves**, podemos indicar **el número mínimo y máximo de veces que un símbolo podrá repetirse**. El primer número del ejemplo es el número 1, y quiere decir que la letra "a" debe aparecer al menos una vez. El segundo número, 4, indica que como máximo puede repetirse cuatro veces.
- **"a{2,}".** También es posible **indicar solo el número mínimo** de veces que un carácter debe aparecer (sin determinar el máximo), haciendo uso de las llaves, indicando el primer número y poniendo la coma (no la olvides).
- **"a{5}".** A diferencia de la forma anterior, **si solo escribimos un número entre llaves**, sin poner la coma detrás, significará que el símbolo debe aparecer **un número exacto de veces**. En este caso, la "a" debe aparecer exactamente 5 veces.
- "[a-z]{1,4}[0-9]+". **Los indicadores de repetición se pueden usar también con corchetes**, dado que los **corchetes representan**, básicamente, **un símbolo**. En el ejemplo anterior se permitiría de una a cuatro letras minúsculas, seguidas de al menos un dígito numérico.

2.2.1.- Expresiones regulares (II).

¿Y cómo uso las expresiones regulares en un programa? Pues de una forma sencilla. Para su uso, Java ofrece las clases **Pattern** y **Matcher** contenidas en el paquete **java.util.regex.***. La clase **Pattern** se utiliza para procesar la expresión regular y "compilarla", lo cual significa verificar que es correcta y dejarla lista para su utilización. La clase **Matcher** sirve para comprobar si una cadena cualquiera sigue o no un patrón.

Veámoslo con un ejemplo:

```
Pattern p=Pattern.compile("[01]+");

Matcher m=p.matcher("00001010");
if (m.matches()) System.out.println("Si, contiene el patrón");
else System.out.println("No, no contiene el patrón");
```

En el ejemplo, el método estático **compile** de la clase **Pattern** permite crear un patrón, dicho método compila la expresión regular pasada por parámetro y genera una instancia de **Pattern** (**p**en el ejemplo). El patrón **p** podrá ser usado múltiples veces para verificar si una cadena coincide o no con el patrón, dicha comprobación se hace invocando el método **matcher**, el cual combina el patrón con la cadena de entrada y genera una instancia de la clase **Matcher**(**m**en el ejemplo). La clase **Matcher** contiene el resultado de la comprobación y ofrece varios métodos para analizar la forma en la que la cadena ha encajado con un patrón:

- **m.matches().** Devolverá true si toda la cadena (de principio a fin) encaja con el patrón o false en caso contrario.
- m.lookingAt(). Devolverá true si el patrón se ha encontrado al principio de la cadena. A diferencia del método matches(), la cadena podrá contener al final caracteres adicionales a los indicados por el patrón, sin que ello suponga un problema.
- m.find(). Devolverá true si el patrón existe en algún lugar la cadena (no necesariamente toda la cadena debe coincidir con el patrón) y false en caso contrario, pudiendo tener más de una coincidencia. Para obtener la posición exacta donde se ha producido la coincidencia con el patrón podemos usar los métodos m.start() y m.end(), para saber la posición inicial y final donde se ha encontrado. Una segunda invocación del método find() irá a la segunda coincidencia (si existe), y así sucesivamente. Podemos reiniciar el método find(), para que vuelva a comenzar por la primera coincidencia, invocando el método m.reset().

Veamos algunas construcciones adicionales que pueden ayudarnos a especificar expresiones regulares más complejas:

- **"[^abc]".** El símbolo **"^",** cuando se **pone justo detrás del corchete de apertura**, significa "**negación**". La expresión regular admitirá cualquier símbolo diferente a los puestos entre corchetes. En este caso, cualquier símbolo diferente de "a", "b" o "c".
- **"^[01]".** Cuando el símbolo **"^"** aparece al **comienzo de la expresión regular**, permite indicar **comienzo de línea o de entrad**a.
- ".". **El punto simboliza cualquier caracter.**
- **"\\d".** Un dígito numérico. Equivale a "[0-9]".
- **"\\D". Cualquier cosa excepto un dígito numérico**. Equivale a "[^0-9]".
- **"\\s". Un espacio en blanco** (incluye tabulaciones, saltos de línea y otras formas de espacio).
- **"\\S". Cualquier cosa excepto un espacio en blanco**.
- **"\\w"**. Cualquier caracter que podrías encontrar en una palabra. Equivale a "[a-zA-Z_0- 9]".

3.- Creación de arrays.

¿Y dónde almacenarías tú la lista de artículos del pedido? Una de las soluciones es usar arrays, puede que sea justo lo que necesita Ana, pero puede que no. Todos los lenguajes de programación permiten el uso de arrays, veamos como son en Java.

Los arrays permiten almacenar una colección de objetos o datos del mismo tipo. Son m útiles y su utilización es muy simple:

- **Declaración del array**. La declaración de un array consiste en decir "esto es un array que sigue la siguiente estructura: "tipo[] nombre;". El tipo será un tipo de variable o una clase existente, de la cual se quieran almacenar varias unidades.

- **Creación del array**. La creación de un array consiste en decir el tamaño que tendrá el array es decir, el número de elementos que contendrá, y se pone de la siguiente forma "nombre=new tipo[dimension]", donde dimensión es un número entero positivo que indic el tamaño del array. Una vez creado el array este no podrá cambiar de tamaño.

Veamos un ejemplo de su uso:

```
int[] n; // Declaración del array.
n = new int[10]; //Creación del array reservando para él un espacio de 10 enteros.
int[] m = new int[10]; // Declaración y creación en un mismo lugar.
```

Una vez hecho esto, ya podemos almacenar valores en cada una de las posiciones del array, usando corchetes e indicando en su interior la posición en la que queremos leer o escribir, teniendo en cuenta que **la primera posición es la cero y la última el tamaño del array menos uno**. En el ejemplo anterior, la primera posición sería la 0 y la última sería la 9.

3.1.- Uso de arrays unidimensionales.

La **modificación** de una posición del array se realiza con una simple asignación.

```
int[] Numeros=new int[3]; // Array de 3 números (posiciones del 0 al 2)

Numeros[0]=99; // Primera posición del array.
Numeros[1]=120; // Segunda posición del array.
Numeros[2]=33; // Tercera y última posición del  array.
```

El **acceso** a un valor ya existente dentro de una posición del array se consigue de forma simil simplemente poniendo el nombre del array y la posición a la cual se quiere acceder en corchetes:

```
int suma=Numeros[0] + Numeros[1] + Numeros[2];
```

Los arrays, como objetos que son en Java, disponen de una propiedad pública muy útil. La propiedad **length** nos permite saber el tamaño de cualquier array.

```
System.out.println("Longitud del array: "+Numeros.length);
```

Paso de parámetros. Para pasar como argumento un array a una función o método, esta debe tener en su definición un parámetro declarado como array. Esto es simplemente que uno de los parámetros de la función sea un array.

```
int sumaArray (int[] j) {
        int suma=0;
        for (int i=0; i<j.length;i++) suma=suma+j[i];
        return suma;
}
```

En el método anterior se pasa como argumento un array numérico, sobre el cual se calcula la suma de todos los números que contiene. Es un uso típico de los arrays, fijate que especificar que argumento es un array es igual que declarar un array, sin la creación del mismo. Para pasar como argumento un array a una función, simplemente se pone el nombre del array:

```
int[] miArray= new int[2];
miArray[0]=3;
miArray[1]=5;
int suma= sumarArray(miArray);
```

En Java las variables se pasan por copia a los métodos, esto quiere decir que cuando se pasa una variable a un método, y se realiza una modificación de su valor en dicho método, el valor de la variable en el método desde el que se ha realizado la invocación no se modifica. Pero cuidado, **eso no pasa con los arrays. Cuando dicha modificación se realiza en un array, es decir, se cambia el valor de uno de los elementos del array, si que cambia su valor de forma definitiva**. Veamos un ejemplo que ilustra ambas cosas:

```
public static void main(String[] args) {
        int j=0;
        int[] i=new int(1);
        i[0]=0;
        modificaArray(j,i);
        System.out.println(j+"-"+i[0]); /* Mostrará por pantalla " modificado en la función, y
                aunque la variable j también dejando el original intacto */
}

int modificaArray(int j, int[] i); {
        j++; int[0]++;          /* Modificación de los valores de la  variable
}
```

3.2.- Inicialización.

Rellenar un array, para su primera utilización, es una tarea muy habitual que puede ser rápidamente simplificada. Vamos a explorar dos sistemas que nos van a permitir inicializar un array de forma cómoda y rápida.

En primer lugar, una forma habitual de crear un array y rellenarlo es simplemente a través de un método que lleve a cabo la creación y el rellenado del array. Esto es sencillo desde que podemos hacer que un método retorne unarray simplemente indicando en la declaración que el valor retornado es tipo[], donde tipo de dato primitivo (**int, short, float**,...) o una clase existente (**String** por ejemplo). Veamos un ejemplo:

```
static int[] ArrayConNumerosConsecutivos (int totalNumeros) {
        int[] r=new int[totalNumeros];
        for (int i=0;i<totalNumeros;i++) r[i]=i;
        return r;
}
```

En el ejemplo anterior se crea un array con una serie de números consecutivos, empezando por el cero, ¿sencillo no? Este uso suele ahorrar bastantes líneas de código en tareas repetitivas.

Otra forma de inicializar los arrays, cuando el número de elementos es fijo y sabido a priori, es indicando entre llaves el listado de valores que tiene el array. En el siguiente ejemplo puedes ver la inicialización de un array de tres números, y la inicialización de un array con tres cadenas de texto:

```
int[] array = {10, 20, 30};
String[] diassemana= {"Lunes", "Martes", "Miércoles", "Jueves",  "Viernes"
```

Pero cuidado, la inicialización solo se puede usar en ciertos casos. La inicialización anterior funciona cuando se trata de un tipo de dato primitivo (**int, short, float, double**, etc.) o un **String**, y algunos pocos casos más, pero no funcionará para cualquier objeto.

Cuando se trata de un **array de objetos**, la inicialización del mismo es un poco más liosa, dado que el **valor inicial de los elementos del array** de objetos será **null**, o **lo que es lo mismo, crear un array de objetos no significa que se han creado las instancias de los objetos**. Hay que crear, para cada posición del array, el objeto del tipo correspondiente con el operador **new**. Veamos un ejemplo con la clase **StringBuilder**.

En el siguiente ejemplo solo aparecerá **null** por pantalla:

```
StringBuilder[] j=new StringBuilderStringBuilder[10];
for (int i=0; i<j.length;i++) System.out.println("Valor" +i +  "="+j[i]);
```

Para solucionar este problema podemos optar por lo siguiente, crear para cada posición del array una instancia del objeto:

```
StringBuilder[] j=new StringBuilder[10];
for (int i=0; i<j.length;i++) j[i]=new StringBuilder("cadena "+i);
```

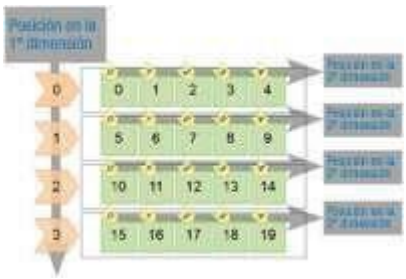
4.- Arrays multidimensionales.

¿Qué estructura de datos utilizarías para almacenar los píxeles de una imagen digital? Normalmente las imágenes son cuadradas así que una de las estructuras más adecuadas es la matriz. En la **matriz** cada valor podría ser el color de cada píxel. Pero, ¿qué es una matriz a nivel de programación? **Pues es un array con dos dimensiones**, o lo que es lo mismo, un array cuyos elementos son arrays de números.

Los arrays multidimensionales están en todos los lenguajes de programación actuales, y obviamente también en Java. La forma de crear un array de dos dimensiones en Java es la siguiente:

```
int[][] a2d=new int[4][5];
```

El código anterior creará un array de dos dimensiones, o lo que es lo mismo, creará un array que contendrá 4 arrays de 5 números cada uno. Veámoslo con un ejemplo gráfico:



Al igual que con los arrays de una sola dimensión, los arrays multidimensionales deben declararse y crearse. Podremos hacer arrays multidimensionales de todas las dimensiones que queramos y de cualquier tipo. En ellos todos los elementos del array serán del mismo tipo, como en el caso de los arrays de una sola dimensión. La declaración comenzará especificando el tipo o la clase de los elementos que forman el array, después pondremos tantos corchetes como dimensiones tenga el array y por último el nombre del array, por ejemplo:

```
int [][][] arrayde3dim;
```

La creación es igualmente usando el operador **new**, seguido del tipo y los corchetes, en los cuales se especifica el tamaño de cada dimensión:

```
arrayde3dim=new int[2][3][4];
```

Todo esto, como ya has visto en un ejemplo anterior, se puede escribir en una única sentencia.

4.1.- Uso de arrays multidimensionales.

¿Y en que se diferencia el uso de un array multidimensional con respecto a uno de una ún dimensión? Pues en muy poco la verdad. Continuaremos con el ejemplo del apartado anterior:

```
int[][] a2d=new int[4][5];
```

Para acceder a cada uno de los elementos del array anterior, habrá que indicar su posición en dos dimensiones, teniendo en cuenta que los índices de cada una de las dimensiones empieza numerarse en 0 y que la última posición es el tamaño de la dimensión en cuestión menos 1.

Puedes asignar un valor a una posición concreta dentro del array, indicando la posición en ca una de las dimensiones entre corchetes:

```
a2d[0][0]=3;
```

Y como es de imaginar, puedes usar un valor almacenado en una posición del array multidimensional simplemente poniendo el nombre del array y los índices del elemento al q deseas acceder entre corchetes, para cada una de las dimensiones del array. Por ejemplo:

```
int suma=a2d[0][0]+a2d[0][1]+a2d[0][2]+a2d[0][3]+a2d[0][4];
```

Como imaginarás, los arrays multidimensionales pueden también ser pasados como parámetro los métodos, simplemente escribiendo la declaración del array en los argumentos del método, forma similar a como se realizaba para arrays de una dimensión. Por ejemplo:

```
static int sumaarray2d(int[][] a2d) { int suma = 0;
        for (int i1 = 0; i1 < a2d.length;  i1++)
            for (int i2 = 0; i2 <  a2d[i1].length; i2++)                    suma += a2d[i1
        return suma;
    }
```

Del código anterior, fijate especialmente en el uso del atributo **length**(que nos permite obtener tamaño de un array). Aplicado directamente sobre el array nos permite saber el tamaño de primera dimensión (**a2d.length**). Como los arrays multidimensionales son arrays que tienen co elementos arrays (excepto el último nivel que ya será del tipo concreto almacenado), para saber tamaño de una dimensión superior tenemos que poner el o los índices entre corchetes seguidos **length**(**a2d[i1].length**).

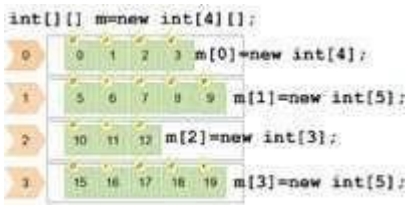
Para saber al tamaño de una segunda dimensión (dentro de una función por ejemplo) hay que hacerlo así y puede resultar un poco engorroso, pero gracias a esto podemos tener array multidimensionales irregulares.

Una matriz es un ejemplo de array multidimensional regular, ¿por qué? Pues porque es un array que contiene arrays de números todos del mismo tamaño.

Cuando esto no es así, es decir, **cuando los arrays de la segunda dimensión son de diferente tamaño entre sí, se puede se puede crear un array irregular** de forma relativamente fácil, veamos un ejemplo para dos dimensiones

- **Declaramos y creamos el array pero sin especificar la segunda dimensión.** Lo que estamos haciendo en realidad es crear simplemente un array que contendrá arrays, sin decir como son de grandes los arrays de la siguiente dimensión:

  int[][] irregular=new int[3][];



- **Después creamos cada uno de los arrays unidimensionales** (del tamaño que queramos) y lo asignamos a la posición correspondiente del array anterior:

  irregular[0]=new int [7]; irregular[1]=new int[15]; irregular[2]=new int[9];

4.2.- Inicialización de arrays multidimensionales.

¿En qué se diferencia la inicialización de arrays unidimensionales de arrays multidimensionales? En muy poco. La inicialización de los arrays multidimensionales es igual que la de los arrays unidimensionales.

Para que una función retorne un array multidimensional, se hace igual que en arrays unidimensionales. Simplemente hay que poner el tipo de dato seguido del número de corchetes correspondiente, según el número de dimensiones del array. Eso claro, hay que ponerlo en la definición del método:

```
int[][] inicializarArray (int n, int m)
{
        int[][] ret=new int[n][m]; for (int
        i=0;i<n;i++)
                for (int j=0;j<m;j++) ret[i][j]=n*m;
        return ret;
}
```

También se puede inicializar un array multidimensional usando las llaves, poniendo después de la declaración del array un símbolo de igual, y encerrado entre llaves los diferentes valores del array separados por comas, con la salvedad de que hay que poner unas llaves nuevas cada vez que haya que poner los datos de una nueva dimensión, lo mejor es verlo con un ejemplo:

```
int[][] a2d={{0,1,2},{3,4,5},{6,7,8},{9,10,11}};
int[][][] a3d={{{0,1},{2,3}},{{0,1},{2,3}}};
```

El primer array anterior sería un array de 4 por 3 y el siguiente sería un array de 2x2x2. Como puedes observar esta notación a partir de 3 dimensiones ya es muy liosa y normalmente no se usa. Utilizando esta notación también podemos inicializar rápidamente arrays irregulares, simplemente poniendo separados por comas los elementos que tiene cada dimensión:

```
int[][] i2d={{0,1},{0,1,2},{0,1,2,3},{0,1,2,3,4},{0,1,2,3,4,5}};
int[][][] i3d={ { {0,1},{0,2} } , {{0,1,3}} , {{0,3,4},{0,1,5} }  };
```

Es posible que en algunos libros y en Internet se refieran a los arrays usando otra terminología. A los arrays unidimensionales es común llamarlos también **arreglos**o **vectores**, a los arrays de dos dimensiones es común llamarlos directamente **matrices**, y a los arrays de más de dos dimensiones es posible que los veas escritos como**matrices multidimensionales**. Sea como sea, lo más normal en la jerga informática es llamarlosarrays, término que procede del inglés.

5.- Clases y métodos genéricos (I).

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

Su objetivo es claro: **facilitar la reutilización del software**, creando métodos y clases que puedan trabajar con diferentes tipos de objetos, evitando incomodas y engorrosas conversiones de tipos.
Veamos un ejemplo sencillo de como transformar un método normal en genérico:

Versiones genérica y no genérica del método de compararTamano.

Versión no genérica	Versión genérica del método

public class util {
public static int compararTamano(Object[] a,Object [] b) {
return a.length- b.length;
} }
	public class util {
	public static <T> int compararTamano (T[] a, T[] b)
	{
	return a.length-b.length;
	}}

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 qué y mayor qué ("**<Integer>**"), justo antes del nombre del método.

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

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

5.1.- Clases y métodos genéricos (II).

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

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

```
public class Util<T> {

    T t1;

    public void metodo(T[] array) {
        //Operaciones sobre el array
}
```

En el ejemplo anterior, la clase **Util** contiene el método invertir cuya función trabajará con el 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:

```
Util<Double> u= new Util<Double>();
```

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 <Double> u) como en la creación (new Util<Double>()).**

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.

6.- Introducción a las colecciones (I).

¿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 ser hacer uso de estos algoritmos.

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(): vacíar la colección.

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

8.- Listas (I).

Amplían el conjunto de operaciones de las colecciones añadiendo operaciones extra, veamos algunas de ellas:

- Las listas 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.
- **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 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**.

8.1.- Listas (II).

Y, ¿cómo se usan las listas? Pues para usar una lista haremos uso de sus implementaciones **LinkedList** y **ArrayList**.

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

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 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 fin Pues será 2, 3 y 5. En el ejemplo cabe destacar el uso del bucle for-each, recuerda que se pue usar en cualquier colección.

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

```
ArrayList<Integer> al=new ArrayList<Integer>(); // Declaración y creación

al.add(10); al.add(11); // Añadimos dos elementos a la lista.

al.set(al.indexOf(11), 12); // Sustituimos el 11 por el 12.
```

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. ejemplo anterior generará un **ArrayList** que contendrá dos números, el 10 y el 12. Veamos ahora 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 lista. Después el método **subList** para extraer una sublista de la lista (que incluía en origen 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 **ArrayList** anterior.

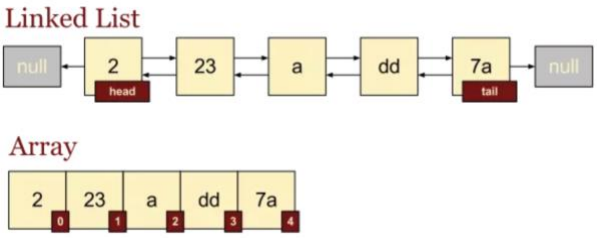
Debes saber que las operaciones aplicadas a una sublista repercuten sobre la lista original. ejemplo, si ejecutamos el método **clear** sobre una sublista, se borrarán todos los elementos de sublista, pero también se borrarán dichos elementos de la lista original.

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

8.2.- Listas (III).

¿Y en qué se diferencia un **LinkedList** de un **ArrayList**? Los **LinkedList** utilizan listas doblemente enlazadas, que son listas enlazadas, pero que permiten ir hacia atrás en la lista de elementos. Los elementos de la lista se encapsulan en los llamados nodos. Los nodos van enlazados unos a otros para no perder el orden y no limitar el tamaño de almacenamiento.

Tener un doble enlace significa que en cada nodo se almacena la información de cuál es el siguiente nodo y además, de cuál es el nodo anterior. Si un nodo no tiene nodo siguiente o nodo anterior, se almacena null o nulo para ambos casos.



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

¿Y esto que quiere decir?

- **Si se vas a realizar muchas operaciones de eliminación de elementos sobre la lista**, conviene usar una lista enlazada (**LinkedList**)
- Si no se van a realizar muchas eliminaciones, **sino que solamente se van a insertar y consultar elementos por posición**, conviene usar una lista basada en arrays redimensionados (**ArrayList**).

LinkedList tiene otras ventajas que nos puede llevar a su uso. Implementa las interfaces **java.util.Queue** y **java.util.Deque**. Dichas interfaces permiten hacer uso de las listas como si fueran una cola de prioridad o una pila, respectivamente.

Las colas, también conocidas como colas de prioridad, son una lista pero que aportan métodos para trabajar de forma diferente. ¿Tú sabes lo que es hacer cola para que te atiendan en una ventanilla? Pues igual. Se trata de que el que primero llega es el primero en ser atendido (FIFO). Simplemente se aportan tres métodos nuevos: meter en el final de la lista (**add** y **offer**), sacar y eliminar el elemento más antiguo (**poll**), y examinar el elemento al principio de la lista sin eliminarlo (**peek**). Dichos métodos están disponibles en las listas enlazadas **LinkedList**:

- boolean add(E e) y boolean offer(E e), retornarán true si se ha podido insertar el elemento al final de la LinkedList.
- E poll() retornará el primer elemento de la LinkedList y lo eliminará de la misma. Al insertar al final, los elementos más antiguos siempre están al principio. Retornará null si la lista está vacía.
- E peek() retornará el primer elemento de la LinkedList pero no lo eliminará, permite examinarlo. Retornará null si la lista está vacía.

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.

Imagínate 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 es 11

Test p2=new Test(12); // Se crea otro objeto Test donde el entero que contiene es 12

LinkedList<Test> lista=new LinkedList<Test>(); // Creamos una lista enlazada

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 de tipo Test
```

¿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).

12.- Tratamiento de documentos estructurados XML.

¿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:

Elementos de un documento 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.	<?xml version="1.0" encoding="ISO-8859-1"?>
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**	<pedido> información del pedido </pedido>

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.	<articulo cantidad="20"> información </articulo>
Texto.	Entre el cierre y la apertura de una etiqueta puede haber texto.	<cliente> Muebles Bonitos S.A. </cliente>
Etiquetas sin contenido.	Cuando una etiqueta no tiene contenido, no tiene porqué 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.	<fecha entrega="1/1/2012" />
Comentario.	Es posible introducir comentarios en XML y estos van dirigidos generalmente a un ser humano que lee directamente el documento XML.	<!-- comentario -->

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>**").

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, atribu ') 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 el estará toda la información del documento XML. Ahora seguro que es más fácil, ¿no?

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:

- XML DOM. Permite transformar un documento XML en un modelo de objetos (de hecho DOM significa Document Object Model), accesible cómodamente desde el lenguaje programación. DOM almacena cada elemento, atributo, texto, comentario, etc. documento XML en una estructura tipo árbol compuesta por nodos fácilmente accesibles, perder la jerarquía del documento. A partir de ahora, la estructura DOM que almacena XML la llamaremos árbol o jerarquía de objetos DOM.

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

- **javax.xml.parsers.DocumentBuilder**: será el procesador y transformará el 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 para el 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 un fabrica de constructor
    DocumentBuilderFactory dbf =  DocumentBuilderFactory.newInstance();
        // 2º A partir de la instancia anterior, fabricamos un construtor
    DocumentBuilder db = dbf.newDocumentBuilder();
        // 3º Procesamos el documento (almacenado en un archivo) y lo asignamos a doc.
    doc=db.parse(CaminoAArchivoXml);
} catch (Exception ex) {
    System.out.println("¡Error! No se ha podido cargar el  documento");
}
```

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? Cómo 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 ("") 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)nl.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). Fijate 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.

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());
                System.out.println("Valor:" + e.getTextContent ());
                break;
        }
    }
```

En el ejemplo anterior se usan varios métodos. El método "**getNodeType()**" la clase **Node** permite saber de que tipo de nodo se trata, generalmen texto (**Node.TEXT_NODE**) o un sub-elemento (**Node.ELEMENT_NODE**). De esta for podremos hacer la conversión de tipos adecuada y gestionar cada elemen según corresponda. También se usa el método "**getTagName**" aplicado a elemento, lo cual permitirá obtener el nombre de la etiqueta, y el método " **getTextContent**" aplicado a un elemento, que permite obtener el texto.

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 un texto a un árbol DOM, primero hay que crear los nodos correspondiente y después insertarlos en la posición que queramos. Para crear un nuevo p de etiquetas o elemento (**Element**) y un nuevo nodo texto (**Text**), lo podemos hacer de la siguiente forma:

```
Element dirTag=doc.createElement("Direccion_entrega")
dirTag.appendChild(doc.createTextNode("C/Pipas n5"));
```

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

```
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 arbol 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.

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/Dirección nueva”);
System.out.println(nuevo.getTextContent());
```