UNIDAD 6: APLICACIÓN DE LAS ESTRUCTURAS DE ALMACENAMIENTO

Módulo Profesional: Programación

ÍNDICE

RESUMEN INTRODUCTORIO

En esta unidad se va a tratar en primer lugar el concepto de array de una dimensión y de más de una dimensión. Posteriormente se tratará el uso no trivial de la clase `String` en Java. Se verá a continuación el tratamiento de archivos XML en Java utilizando DOM y SAX. Finalizaremos revisando una de las estructuras dinámicas de datos más usadas como son las listas. Para cada uno de los apartados veremos ejemplos en el lenguaje de programación Java.

INTRODUCCIÓN

Una estructura de datos es una forma de organizar una serie de datos, ya sean simples o compuestos. En Java tenemos estructuras de datos "estáticas" como pueden ser los arrays y estructuras de datos "dinámicas" como son las colecciones y los mapas. Dentro de las colecciones en Java podemos encontrar las listas. En cuanto al tratamiento de los archivos XML en Java, nos presentan dos posibles formas de parsearlos (o interpretarlos) que son SAX y DOM. El almacenamiento de cadenas de texto en algunos lenguajes se realiza de una forma trivial. En Java se pueden usar clases como String, StringBuffer o StringBuilder, y su funcionamiento no es tan trivial.

CASO INTRODUCTORIO

La aplicación que estamos desarrollando en nuestra empresa maneja una gran cantidad de información. Hemos analizado los requisitos y hemos llegado a la conclusión de que antes de almacenar toda esta información en una base de datos, vamos a utilizar algunas estructuras de datos para organizarla. Utilizaremos no solo estructuras de datos, sino también algunos archivos con información en formato XML que tendremos que parsear.

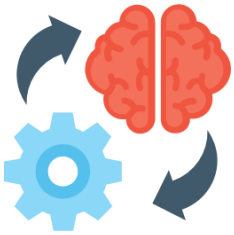
Al finalizar la unidad el alumnado:

- Conocerá y aplicará las estructuras de almacenamiento de datos.
- Será capaz de programar en Java dichas estructuras y realizar operaciones básicas con ellas.
- Conocerá el tratamiento que hace Java de XML y la forma de implementarlo.

1. ARRAYS

El array es una de las estructuras de datos más ampliamente utilizada por su flexibilidad para derivar en complejas estructuras de datos y su simplicidad. Empezaremos con una definición: *un **array** es una secuencia de elementos, donde cada **elemento** (un grupo de bytes de memoria que almacenan un único ítem de datos) se asocia con al menos un **índice** (entero no negativo).* Esta definición da lugar a cuatro puntos interesantes:

- Cada elemento ocupa el **mismo número de bytes**; el número exacto depende del tipo de datos del elemento.
- Todos los elementos son del **mismo tipo**.
- Se tiende a pensar que los elementos de un array ocupan **localizaciones de memoria consecutivas**. Cuando se vean los arrays bidimensionales se descubrirá que **no siempre es así**.
- El número de índices asociados con cada elemento es la ***dimensión*** del array.



RECUERDA

Esta sección se enfoca exclusivamente en arrays de una y dos dimensiones porque los arrays de más dimensiones no se utilizan de forma tan frecuente.

ENLACE DE INTERÉS

Para ampliar información sobre qué es un array y cómo se utilizan puede visitar el siguiente enlace donde además encontrará ejemplos en Java.

- Los arrays

2. ARRAYS DE UNA DIMENSIÓN

El tipo de array más simple tiene una sola dimensión: cada elemento se asocia con un único índice. El lenguaje Java proporciona varias técnicas para crear un array de una dimensión:

- **Utilizar sólo un Inicializador**
  Utilizando un inicializador se puede utilizar cualquiera de estas dos sintaxis:

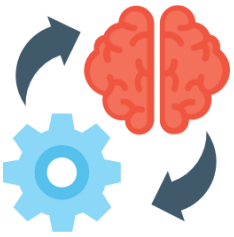
```
type variable_name '[' ']' [ '=' initializer ] ';'
type '[' ']' variable_name [ '=' initializer ] ';'
```

  Donde el inicializador tiene la siguiente sintaxis:

```
'{' initial_value1 ',' initial_value2 ',' ... '}'
```

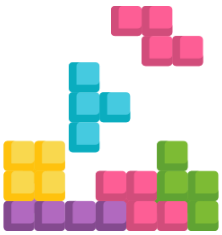
  El siguiente fragmento ilustra cómo crear un array de animales:

```
String animales[] = { "Tigre", "Cebra", "Canguro" };
```



RECUERDA

Los desarrolladores Java normalmente sitúan los corchetes cuadrados después del tipo (int[] test_scores) en vez de escribirlos después del nombre de la variable (int test_scores[]) cuando declaran una variable array. Mantener toda la información del tipo en un único lugar mejora la lectura del código.



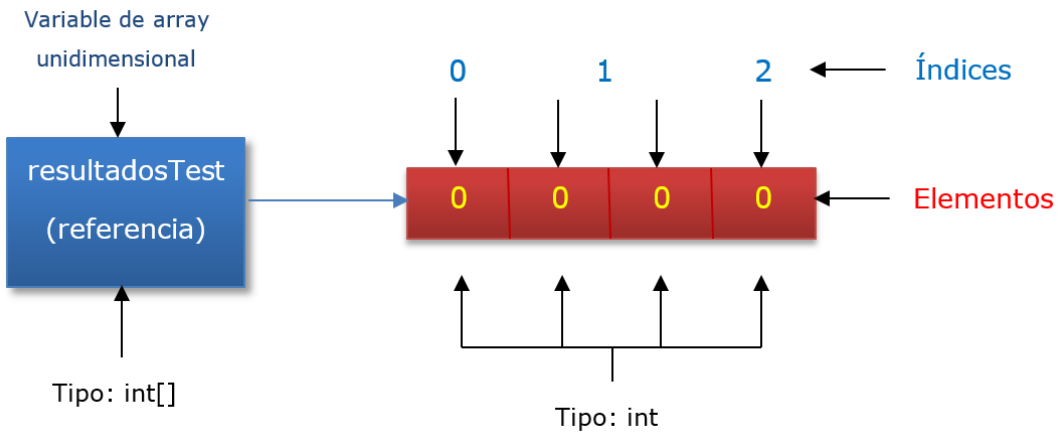
EJEMPLO PRÁCTICO

El siguiente fragmento de código utiliza sólo la palabra clave new para crear un array uni-dimensional que almacena datos de un tipo primitivo:

```
int[] resultadosTest = new int[4];
```

int[] resultadosTest declara una variable array unidimensional (resultadosTest) junto con su tipo de variable (int[]). El tipo de referencia int[] significa que cada elemento debe contener un ítem del tipo primitivo entero. **new int[4]** crea un array unidimensional asignando memoria para cuatro elementos enteros consecutivos. Cada elemento contiene un único entero y se inicializa a cero. El operador *igual a* (=) asigna la referencia del array unidimensional a resultadosTest.

La siguiente figura ilustra los elementos y la variable array unidimensional resultante:



- **Utilizar sólo la Palabra Clave "new"**
  Utilizando la palabra clave new se puede utilizar cualquiera de estas dos sintaxis:

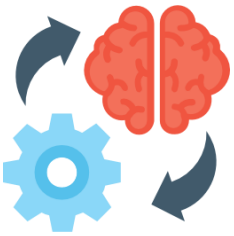
```
type variable_name '[' ']' '=' 'new' type '[' integer_expression ']' ';'
type '[' ']' variable_name '=' 'new' type '[' integer_expression ']' ';'
```

Para ambas sintaxis:

- **variable_name** especifica el nombre de la variable del array unidimensional.
- **Type** especifica el tipo de cada elemento. Como la variable del array unidimensional contiene una referencia a un array unidimensional, el tipo es type[].

- **La palabra clave** `new` seguida **por** `type` y seguida por `integer_expression` entre corchetes cuadrados ([]) especifica el número de elementos. `New` asigna la memoria para los elementos del array unidimensional y pone ceros en todos los bits de los bytes de cada elemento, lo que significa que cada elemento contiene un valor por defecto que se interpreta según su tipo.
- `=` asigna la referencia al array unidimensional a la variable `variable_name`.

Los arrays unidimensionales de tipos primitivos almacenan datos que son valores primitivos. Por el contrario, los arrays unidimensionales del tipo referencia almacenan datos que son referencias a objetos.

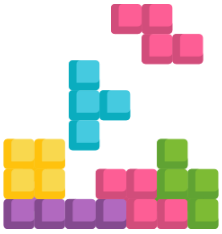


RECUERDA

Cuando se crea un array unidimensional basado en un tipo primitivo, el compilador requiere que aparezca la palabra clave que indica el tipo primitivo en los dos lados del operador igual-a. De otro modo, el compilador lanzará un error. Por ejemplo:

```
int[] resultadosTest= new long[20];
```

Es ilegal porque las palabras claves int y long representan tipos primitivos incompatibles.



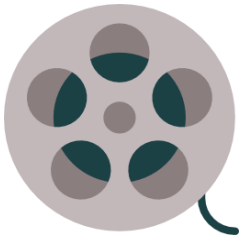
EJEMPLO PRÁCTICO

El siguiente fragmento de código utiliza la palabra clave new para crear un par de arrays unidimensionales que almacenan datos basados en tipo referencia:

```
Reloj[] c1 = new Reloj[3];
Reloj[] c2 = new RelojAlarma[3];
```

`Reloj [] c1 = new Reloj [3];` declara una variable array unidimensional, `c1` de tipo `Reloj[]`, asigna memoria para un array unidimensional `Reloj` que consta de tres elementos consecutivos, y asigna la referencia del array `Reloj` a `c1`. Cada elemento debe contener una referencia a un objeto `Reloj` (asumiendo que Reloj es una clase concreta) o un objeto creado desde una subclase de Reloj y lo inicializa a null.

`Reloj [] c2 = new RelojAlarma [3];` se asemeja a la declaración anterior, excepto en que se crea un array unidimensional `RelojAlarma`, y su referencia se asigna a la variable `Reloj[]` de nombre `c2`. (Asume `RelojAlarma` como subclase de `Reloj`).



VIDEO DE INTERÉS

En el siguiente vídeo se explica cómo utilizar los arrays en Java:

- Tutorial Creación y manejo de vectores o arrays en java

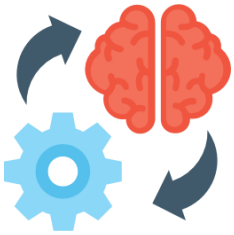
- **Utilizar la palabra clave "new" y un Inicializador**
  Utilizar la palabra clave `new` con un inicializador requiere la utilización de alguna de las siguientes sintaxis:

```
type variable_name> '[' ']' '='  'new' type '[' ']' initializer ';'
type '[' ']' variable_name '='  'new' type '[' ']' initializer ';'
```

Donde `initializer` tiene la siguiente sintaxis:

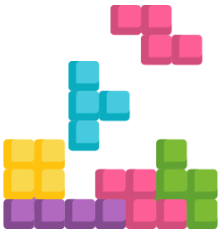
```
'{' [ initial_value [ ',' ... ] ] '}'
```

- `variable_name` especifica el nombre de la variable del array unidimensional.
- `type` especifica el tipo de cada elemento. Como la variable del array unidimensional contiene una referencia a un array unidimensional, el tipo es `type[]`.
- La palabra clave `new` seguida por `type` y seguida por corchetes cuadrados ([]) vacíos, seguido por `initializer`. No se necesita especificar el número de elementos entre los corchetes cuadrados porque el compilador cuenta el número de entradas en el inicializador. `new` asigna la memoria para los elementos del array unidimensional y asigna cada una de las entradas del inicializador a un elemento en orden de izquierda a derecha.
- `=` asigna la referencia al array unidimensional a la variable `variable_name`.



RECUERDA

Un array unidimensional (o de más dimensiones) creado con la palabra clave new con un inicializador algunas veces es conocido como un array anónimo.



EJEMPLO PRÁCTICO

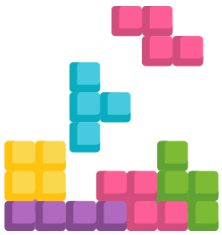
El siguiente fragmento de código utiliza la palabra clave new con un inicializador para crear un array unidimensional con datos basados en tipos primitivos:

```
int[] resultadosTest = new int[] { 70, 80, 20, 30 };
```

`int[] resultadosTest` declara una variable de array unidimensional `resultadosTest` junto con su tipo de variable (`int []`). El código `new int[] { 70, 80, 20, 30 }` crea un array unidimensional asignando memoria para cuatro elementos enteros consecutivos; y almacena `70` en el primer elemento, `80` en el segundo, `20` en el tercero, y `30` en el cuarto. La referencia del array unidimensional se asigna a `resultadosTest`.

No especifique una expresión entera entre los corchetes cuadrados del lado derecho de la igualdad. De lo contrario, el compilador lanzará un error. Por ejemplo, new int[3] {70, 80, 20, 30} hace que el compilador lance un error porque puede determinar el número de elementos partiendo del inicializador. Además, la discrepancia está entre el número 3 que hay en los corchetes y las cuatro entradas que hay en el inicializador.

La técnica de crear arrays unidimensionales con la palabra clave `new` y un inicializador también soporta la creación de arrays que contienen referencias a objetos.



EJEMPLO PRÁCTICO

El siguiente fragmento de código utiliza esta técnica para crear una pareja de arrays unidimensionales que almacenan datos del tipo referencia:

```
Reloj[] c1 = new Reloj[3] { new Reloj() };
Reloj[] c2 = new RelojAlarma[3] { new RelojAlarma() };
```

`Reloj[] c1 = new Reloj[3];` declara una variable de array unidimensional `c1` del tipo `Reloj[]`, asigna memoria para un array `Reloj` que consta de un solo elemento, crea un objeto `Reloj` y asigna su referencia a este elemento, y asigna la referencia del array `Reloj` a `c1`. El código `Reloj[] c2 = new RelojAlarma[3];` se parece a la declaración anterior, excepto en que crea un array unidimensional de un sólo elemento `RelojAlarma` que inicializa un objeto del tipo `RelojAlarma`.

2.1 Trabajar con un array unidimensional

Después de crear un array unidimensional, hay que almacenar y recuperar datos de sus elementos. Con la siguiente sintaxis se realiza esta tarea:

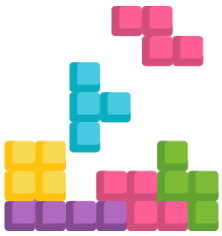
```
variable_name '[' integer_expression ']'
```

`integer_expression` identifica un índice de elemento y debe evaluarse como un entero entre 0 y uno menos que la longitud del array unidimensional (que devuelve `variable_name.length`). Un índice menor que 0 o mayor o igual que la longitud causa que se lance una excepción del tipo `ArrayIndexOutOfBoundsException`.

ENLACE DE INTERÉS

En este enlace se encuentra un resumen de arrays unidimensionales en Java.

- Repaso arrays o arreglos unidimensionales en Java

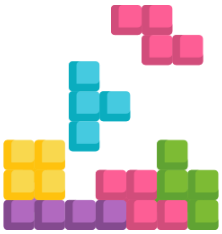


EJEMPLO PRÁCTICO

El siguiente fragmento de código ilustra accesos legales e ilegales a un elemento:

```
public static void main(String[] args) {
	String [] meses = new String [] { "Ene", "Feb", "Mar", "Abr",
		"May", "Jun","Jul", "Ago", "Sep", "Oct", "Nov", "Dic" };
	System.out.println (meses [0]); // Salida: Ene
	// La siguiente instrucción provoca una excepción del tipo
	// ArrayIndexOutOfBoundsException porque el índice es mayor
	// que la longitud del array menos uno
	System.out.println (meses [meses.length]);
	System.out.println (meses [meses.length - 1]); // Salida: Dic
	// La siguiente instrucción provoca una excepción del tipo
	// ArrayIndexOutOfBoundsException porque el índice es < que 0
	System.out.println (meses [-1]);
}
```

Ocurre una situación interesante cuando se asigna la referencia de una subclase de un array a una variable de array de la superclase, porque un subtipo de array es una clase del supertipo de array. Si intenta asignar una referencia de un objeto de la superclase a los elementos del array de la subclase, se lanza una excepción del tipo `ArrayStoreException`.



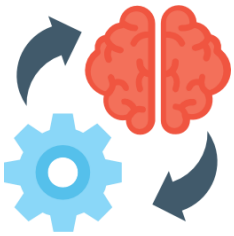
EJEMPLO PRÁCTICO

El siguiente fragmento demuestra esto:

```
public static void main(String[] args) {
	RelojAlarma[] ac = new RelojAlarma[1];
	Reloj[] c = ac;
	c[0] = new Reloj();
}
```

El compilador no dará ningún error porque todas las líneas son legales. Sin embargo, durante la ejecución, `c[0] = new Reloj();` resulta que lanza una excepción del tipo `ArrayStoreException`. Esta excepción ocurre porque se puede intentar acceder a un miembro específico de `RelojAlarma` mediante una referencia a un objeto `Reloj`.

Por ejemplo, supongamos que `RelojAlarma` contiene un método `public void suenaAlarma()`, `Reloj` no lo tiene, y el fragmento de código anterior se ejecuta sin lanzar una excepción del tipo `ArrayStoreException`. Un intento de ejecutar a `c[0].suenaAlarma();` bloquea la JVM *Máquina Virtual Java* porque estamos intentando ejecutar este método en el contexto de un objeto `Reloj` (que no incorpora un método `suenaAlarma()`).



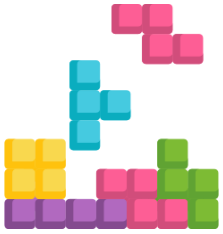
RECUERDA

Tenga cuidado cuando acceda a los elementos de un array porque podría recibir una ArrayIndexOutOfBoundsException o una ArrayStoreException.

2.2 Algoritmos de búsqueda-lineal, búsqueda-binaria y ordenación de burbuja

Los desarrolladores normalmente escriben código para buscar datos en un array y para ordenar ese array. Hay tres algoritmos muy comunes que se utilizan para realizar estas tareas.

- **Búsqueda Lineal:** El algoritmo de búsqueda lineal busca en un array unidimensional un dato específico. La búsqueda primero examina el elemento con el índice 0 y continúa examinando los elementos sucesivos hasta que se encuentra el ítem o hasta que no quedan más elementos que examinar.
  Dos de las ventajas de la búsqueda lineal son la **simplicidad** y la habilidad de buscar tanto arrays **ordenados** como **desordenados**. Su única **desventaja** es el **tiempo** empleado en examinar los elementos. El número medio de elementos examinados es la **mitad de la longitud del array**, y el máximo número de elementos a examinar es la longitud completa. Por ejemplo, un array unidimensional con 20 millones de elementos requiere que una búsqueda lineal examine una media de 10 millones de elementos y un máximo de 20 millones.
  Como este tiempo de examen podría afectar seriamente al rendimiento, utilice la búsqueda lineal para arrays unidimensionales con relativamente **pocos elementos**.



EJEMPLO PRÁCTICO

El siguiente código Java demuestra este algoritmo:

```
public static void main(String[] args) {
	int i=0;
	int buscado = 72;
	int x[] = {20, 15, 12, 30, -5, 72, 456};
	boolean encontrado=false;
	while (!encontrado) {
		if (x[i] == buscado) {
			System.out.println("Encontrado: " + x[i]);
			encontrado=true;
		}
		i++;
	}
	if (!encontrado)
		System.out.println("No encontrado: " + buscado);
}
```

La salida será:

```
Encontrado: 72
```

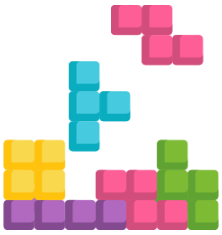
ARTÍCULO DE INTERÉS

En el siguiente artículo se resume de manera clara y concisa la búsqueda lineal en arrays utilizando el lenguaje de programación Java. Se describen las condiciones, además de explicar los algoritmos antes de codificarlos en Java. Muy interesante para entender cómo funciona la búsqueda:

- Algoritmos de búsqueda

- **Búsqueda Binaria:** Al igual que en la búsqueda lineal, el algoritmo de búsqueda binaria busca un dato determinado en un array unidimensional. Sin embargo, al contrario que la búsqueda lineal, la búsqueda binaria **divide** el array en sección inferior y superior calculando el índice central del array. Si el dato se encuentra en ese elemento, la búsqueda binaria termina.

Si el dato es numéricamente menor que el dato del elemento central, la búsqueda binaria calcula el índice central de la mitad inferior del array, ignorando la sección superior y repite el proceso. La búsqueda continúa hasta que se encuentre el dato o se exceda el límite de la sección (lo que indica que el dato **no existe** en el array).
La **única ventaja** de la búsqueda binaria es que **reduce el tiempo** empleado en examinar elementos. El número máximo de elementos a examinar es $\log_2 n$ (donde n es la longitud del array unidimensional). Por ejemplo, un array unidimensional con 1.048.576 elementos requiere que la búsqueda binaria examine un máximo de 20 elementos. La búsqueda binaria tiene dos **inconvenientes**; el **incremento de complejidad** y la necesidad de **preordenar** el array.

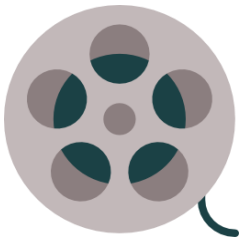


EJEMPLO PRÁCTICO

El siguiente código representa el equivalente Java del pseudocódigo anterior:

```
public static void main(String[] args) {
      int[] x = { -5, 12, 15, 20, 30, 72, 456 };
      int indiceInferior = 0;
      int indiceSuperior = x.length-1;
      int indiceMedio;
      int buscado = 72;
      boolean fin=false;
      while ((indiceInferior <= indiceSuperior) && (!fin)) {
         indiceMedio = (indiceInferior + indiceSuperior) / 2;
         if (buscado > x[indiceMedio])
             indiceInferior = indiceMedio + 1;
         else if (buscado < x[indiceMedio])
             indiceSuperior = indiceMedio - 1;
         else
              fin=true;
      }
      if (indiceInferior>indiceSuperior)
        System.out.println (buscado + " no encontrado");
      else
        System.out.println (buscado + " encontrado");
}
```

- **Ordenación de Burbuja:** Cuando entra en juego la ordenación de datos, la ordenación de burbuja es uno de los algoritmos **más simples**. Este algoritmo hace varios pases sobre un array unidimensional. Por cada pase, el algoritmo compara datos adyacentes para determinar si numéricamente es mayor o menor. Si el dato es mayor (para ordenaciones ascendentes) o menor (para ordenaciones descendientes) los datos se intercambian y se baja de nuevo por el array. En el último pase, el dato mayor (o menor) se ha movido al final del array. Este efecto "burbuja" es el origen de su nombre.



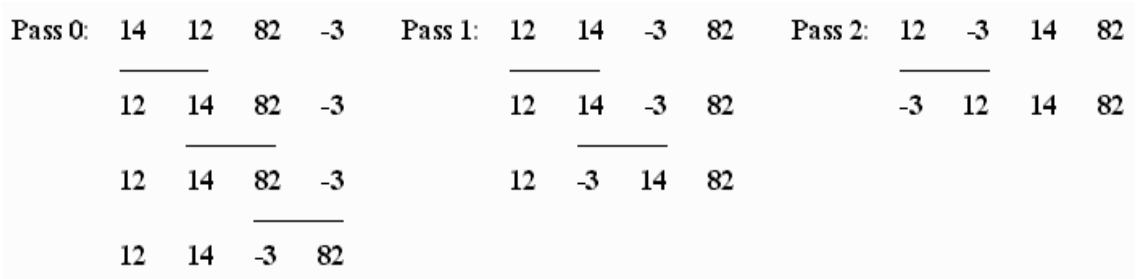
VIDEO DE INTERÉS

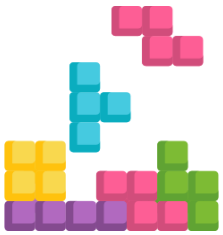
Otro de los algoritmos utilizados para la búsqueda sería el binario. Para ampliar información sobre éste visite el siguiente enlace:

- Método de búsqueda binario

La siguiente figura muestra una ordenación de burbuja ascendente de un array unidimensional de cuatro elementos. Hay tres pasos, el paso 0 realiza tres comparaciones y dos intercambios, el paso 1 realiza dos comparaciones y un intercambio y el paso 2 realiza una comparación y un intercambio.

Aunque la ordenación de burbuja es uno de los algoritmos de ordenación más simples, también es uno de los más **lentos**. Entre los algoritmos más **rápidos** se incluyen la **ordenación rápida** y la **ordenación de pila**.





EJEMPLO PRÁCTICO

El siguiente listado presenta el algoritmo programado en Java:

```
public static void main(String[] args) {

            int[] x= {20,15,12,30,-5,72,456};
            int i, j, aux;
              for (i = 0; i < x.length-1; i++) {
                  for (j = 0; j < x.length - i - 1; j++) {
                      if (x[j + 1] < x[j]) {
                          aux = x[j + 1];
                          x[j + 1] = x[j];
                          x[j] = aux;
                      }
                  }

              for (i=0;i<x.length;i++) {
                          System.out.println("Valor del array " +x[i]);
            }
            }
```

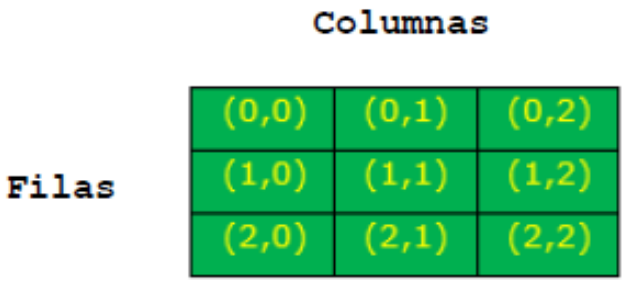
SABÍAS QUE...

Otro algoritmo muy utilizado para arrays unidimensionales copia los elementos de un array fuente en otro array de destino. En vez de escribir su propio código para realizar esta terea puede utilizar el método `public static void array copy(Object src, int srcindex, Object dst, int dstindex, int length)` de la clase `java.lang.System`, que es la forma más rápida de realizar la copia.

3. ARRAYS DE DOS DIMENSIONES

Un array de dos dimensiones, también conocido como ***tabla*** o ***matriz***, donde cada elemento se asocia con una **pareja de índices**, es otro array simple. Se conceptualiza un array bidimensional como una **cuadrícula rectangular** de elementos divididos en **filas** y **columnas**, y se utiliza la notación *(fila, columna)* para identificar a un elemento específico.

La siguiente figura ilustra esta visión conceptual y la notación específica de los elementos:



Java proporciona tres técnicas para crear un array bidimensional:

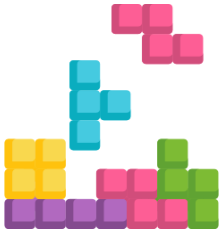
- **Utilizar sólo un Inicializador**
  Esta técnica requiere una de estas sintaxis:

```
type variable_name '[' ']' '[' ']' '='   '{' [ rowInitializer [ ',' ... ] ]
'}' ';'
type '[' ']' '[' ']' variable_name '='  '{' [rowInitializer [ ',' ... ] ] '}'
';'
```

  Donde *rowInitializer* tiene la siguiente sintaxis:

```
'{' [ initial_value [ ',' ... ] ] '}'
```

  Para ambas sintaxis:
  - ***variable_name*** especifica el nombre de la variable del array bidimensional.
  - ***type*** especifica el tipo de cada elemento. Como una variable de array bidimensional contiene una referencia a un array bidimensional, su tipo es `type[ ][ ]`.
  - Especifica cero o más inicializadores de filas entre los corchetes (`{ }`). Si no hay inicializadores de filas, el array bidimensional está vacío. Cada inicializador de fila especifica **cero o más valores iniciales** para las entradas de las columnas de esa fila. Si no se especifican valores para esa fila, la fila está **vacía**.
  - `=` se utiliza para asignar la referencia del array bidimensional a `variable_name`.



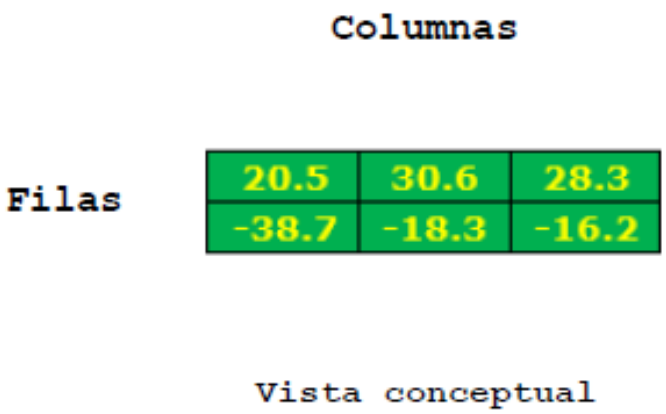
EJEMPLO PRÁCTICO

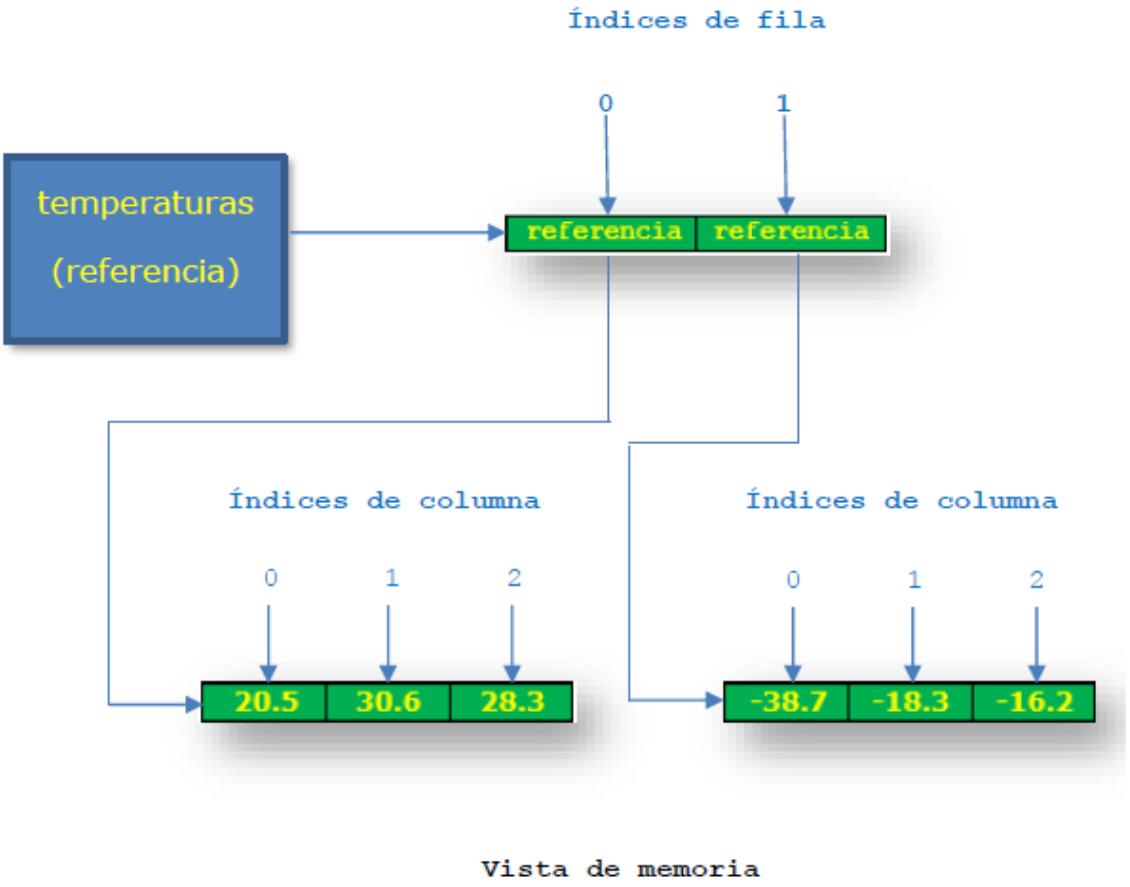
El siguiente código usa sólo un inicializador para crear un array bidimensional que almacena datos basados en un tipo primitivo:

```
double[][] temperaturas = {{20.5,30.6,28.3},
{-38.7,-18.3,-16.2 }}; // Temperaturas Celsius
```

double[][] temperaturas declara una variable de array bidimensional (temperaturas) junto con su tipo de variable (double[][]). El tipo de referencia double[][] significa que cada elemento debe contener datos del tipo primitivo double. **{{20.5,30.6,28.3},{-38.7,-18.3,-16.2 }}** especifica un array bidimensional de **dos filas** por **tres columnas**, donde la primera fila contiene los datos 20.5, 30.6, y 28.3, y la segunda fila contiene los datos -38.7, -18.3, y -16.2. El operador igual-a (=) asigna la referencia del array bidimensional a temperaturas.

La siguiente figura ilustra el array bidimensional resultante desde un punto de vista conceptual y de memoria.





Vista de memoria

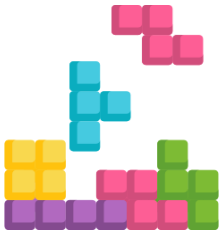
- **Utilizar sólo la palabra clave "new"**
  Esta técnica requiere cualquiera de estas sintaxis:

```
type variable_name '[' ']' '[' ']' '=' 'new' type '[' integer_expression ']'
'[' ']' ';'

type '[' ']' '[' ']' variable_name '=' 'new' type '[' integer_expression ']'
'[' ']' ';'
```

  En ambas sintaxis:
  - ***variable_name*** especifica el nombre de la variable del array bidimensional.
  - ***type*** especifica el tipo de cada elemento. Como es una variable de array bidimensional contiene una referencia a un array bidimensional, su tipo es `type[ ][ ]`.
  - La palabra clave **`new`** **seguida por** **`type`** y por una **expresión entera entre corchetes cuadrados**, que identifica el número de filas. La instrucción `new` asigna memoria para las filas del array unidimensional de filas y pone a cero todos los bytes de cada elemento, lo que significa que cada elemento contiene una referencia nula.

Debe crear un array unidimensional de columnas separado y asignarle su referencia cada elemento fila.

- `=` se utiliza para asignar la referencia del array bidimensional a `variable_name`.



EJEMPLO PRÁCTICO

El siguiente fragmento de código usa sólo la palabra clave new para crear un array bidimensional que almacena datos basados en un tipo primitivo:

```
public static void main(String[] args) {
        double[][] temperaturas = new double[2][];
        // La matriz tiene 2 filas
        temperaturas[0]=new double[3];
        // La primera fila tiene 3 columnas
        temperaturas[1]=new double[3];
        // La segunda fila tiene 3 columnas
        temperaturas[0][0]=20.5; // Se asignan valores para la fila 0
        temperaturas[0][1]=30.6;
        temperaturas[0][2]=28.3;
        temperaturas[1][0]=-38.7; //Se asignan valores para la fila 1
        temperaturas[1][1]=-18.3;
        temperaturas[1][2]=-16.2;
}
```

- **Utilizar la palabra clave "new" y un inicializador**
  Esta técnica requiere una de estas sintaxis:

```
type variable_name '[' ']' '[' ']' '='
'new' type '[' ']' '[' ']' '{' [ rowInitializer [ ',' ... ] ] '}' ';'
type '[' ']' '[' ']' variable_name '='
'new' type '[' ']' '[' ']' '{' [ rowInitializer [ ',' ... ] ] '}' ';'
```

Donde *rowInitializer* tiene la siguiente sintaxis:

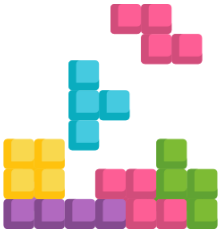
```
'{' [ initial_value [ ',' ... ] ] '}'
```

En ambas sintaxis:

- ***variable_name*** especifica el nombre de la variable del array bidimensional.
- ***type*** especifica el tipo de cada elemento. Como es una variable de array bidimensional contiene una referencia a un array bidimensional, su tipo es `type[ ][ ]`.
- La palabra clave ***new*** seguida por ***type*** y por dos parejas de

corchetes cuadrados vacíos, y cero o más inicializadores de filas entre un par de corchetes cuadrados. Si no se especifica ningún inicializador de fila, el array bidimensional está vacío. Cada inicializador de fila especifica cero o más valores iniciales para las columnas de esa fila.

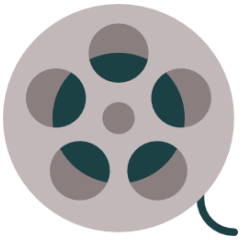
- `=` se utiliza para asignar la referencia del array bidimensional a `variable_name`.



EJEMPLO PRÁCTICO

El siguiente fragmento de código usa la palabra clave `new` y un inicializador para crear un array bidimensional que almacena datos basados en un tipo primitivo:

```
double[ ][ ] temperaturas = new double[ ][ ] { {20.5,30.6,28.3},
      {-38.7,-18.3,-16.2 } };
```



VIDEO DE INTERÉS

Los arrays multidimensionales son más difíciles de entender. Por esto se recomienda visitar el siguiente vídeo para ver cómo trabajar con matrices en la práctica:

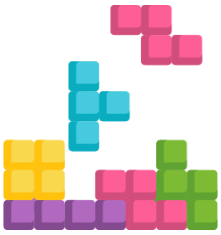
- Arrays de dos dimensiones - Español

3.1 Trabajar con arrays bidimensionales

Después de crear un array bidimensional, se querrá almacenar y recuperar datos de sus elementos. Se puede realizar esta tarea con la siguiente sintaxis:

```
variable_name '[' integer_expression1 ']' '[' integer_expression2 ']'
```

`integer_expression1` identifica el índice de fila del elemento y va de cero hasta la longitud del array menos uno. Igual ocurre con `integer_expression2`, pero para el índice de columna. Como todas las filas tienen la misma longitud, se puede encontrar conveniente especificar `variable_name[0].length` para especificar el número de columnas de cualquier fila.



EJEMPLO PRÁCTICO

El siguiente fragmento de código almacena y recupera datos de un elemento de un array bidimensional:

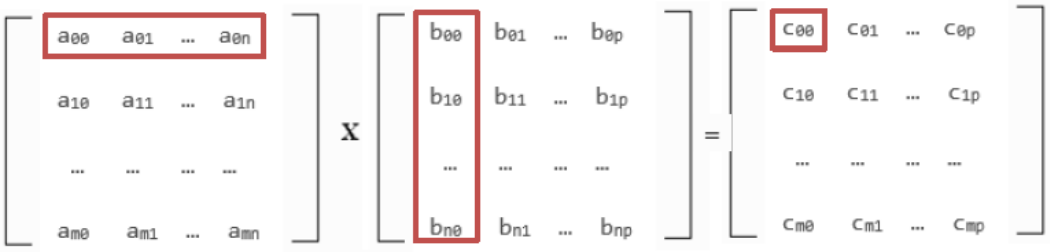
```
public static void main(String[] args) {
      double[][] temperaturas = { {20.5,30.6,28.3},
                                  {-38.7,-18.3,-16.2} };
      temperaturas[0][1] = 18.3; // Reemplaza 30.6 por 18.3
      System.out.println(temperaturas[1][2]); // Salida: -16.2
 }
```

3.2 Algoritmo de multiplicación de matrices

Multiplicar una matriz por otra es una operación común en el trabajo con gráficos, con datos económicos, o con datos industriales. Los desarrolladores normalmente utilizan el algoritmo de multiplicación de matrices para completar esa multiplicación. ¿Cómo funciona ese algoritmo? Dejemos que 'A' represente una matriz con 'm' filas y 'n' columnas. De forma similar, 'B' representa un matriz con 'p' filas y 'n' columnas.

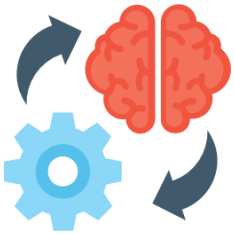
Multiplicar A por B produce una matriz C obtenida de multiplicar todas las entradas de A por su correspondencia en B.

La siguiente figura ilustra estas operaciones:



$$c_{00} = a_{00}b_{00} + a_{01}b_{10} + \ldots + a_{0n}b_{n0}$$

Se requieren tres bucles `FOR` para realizar la multiplicación. El bucle más interno multiplica un elemento de la fila de la matriz `a` por un elemento de la columna de la matriz `b` y añade el resultado a una sola entrada de la matriz `result`.



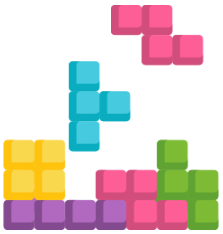
RECUERDA

La multiplicación de matrices requiere que el **número de columnas** de la matriz de la izquierda (A) sea igual al **número de filas** de la matriz de la derecha (B). Por ejemplo, para multiplicar una matriz A de cuatro columnas por fila por una matriz B (como en A x B), B debe contener exactamente cuatro filas.

ENLACE DE INTERÉS

Para entender cómo multiplicar matrices en Java es preciso tener claro en qué consiste este proceso matemáticamente hablando. Visite el siguiente enlace para entenderlo mejor:

- Matemáticas/Matrices/Multiplicación



EJEMPLO PRÁCTICO

El siguiente método realiza la multiplicación de matrices en Java:

```
static int[][] multiplicaMatrices(int[][] a, int[][] b) {
    // Si el nº de columnas de la matriz a es distinto
    // del nº de filas de la matriz b nos muestra un error
    if (a[0].length != b.length) {
          System.err.println("El nº de columnas de la matriz a es \r\n" +
                             " distinto del nº de filas de la matriz b ");
          return null;
    }
    int[][] result = new int[a.length][];
    for (int i=0; i<result.length;i++)
          result[i]=new int[b[0].length];
    // Realiza la multiplicación y la suma
    for (int i=0;i<a.length;i++)
          for (int j=0;j<b[0].length;j++)
                for (int k=0;k<a[0].length;k++)
                      result[i][j] += a[i][k] * b[k][j];
    // Devuelve la matriz resultante
    return result;
}
```

4. CLASE STRING: REPRESENTANDO UNA CADENA

Una cadena de texto no deja de ser más que la sucesión de un conjunto de caracteres alfanuméricos, signos de puntuación y espacios en blanco con más o menos sentido.

Se puede encontrar desde la archiconocida cadena "Hola Mundo" y la no menos "Mi primera cadena de texto", pasando por las cadenas de texto personalizadas "Pepe", "Pepe López", ... hasta las inclasificables "asdf".

Todas ellas serán representadas en java con la clase **String.** Para encontrar la clase **String** dentro de las librerías de Java tendremos que ir a **java.lang.String.**

ENLACE DE INTERÉS

El siguiente enlace dirige a la web de Oracle donde podrá encontrar toda la información sobre la clase String:

- Clase String

4.1 Creación de una cadena

Para crear una cadena existen dos opciones:

Instanciar la clase String, que sería una **creación explicita** de la clase

```
String sMiCadena = new String("Cadena de Texto");
```

Crear implícitamente la cadena de texto. Es decir, simplemente se le asigna el valor al objeto.

```
String sMiCadena = "Cadena de Texto";
```

En este caso, Java, creará un objeto String para tratar esta cadena.

4.2 Crear una cadena vacía

Se puede tener la necesidad de crear una cadena vacía. Puede darse el caso de que no siempre se sepa lo que se va a poner de antemano en la cadena de texto. Muchas veces la cadena de texto la proporcionará el usuario, otro sistema,...

Para poder crear la cadena vacía bastará con asignarle el valor "", o bien, utilizar el constructor vacío.

```
String sMiCadena = "";
```

```
String sMiCadena = new String();
```

- **Constructores String:** Se puede resumir que se tienen dos tipos de constructores principales de la clase String:
  - **`String()`**, que construirá un objeto `String` sin inicializar.
  - **`String(String original)`**, construye una clase `String` con otra clase `String` que recibirá como argumento.
- **Volcado de una cadena de texto a la consola:** Solo quedará saber cómo volcar una cadena por pantalla. Esto se hará con la clase `System.out.println` que recibirá como parámetro el objeto `String`. Por ejemplo:

```
System.out.println("Mi Cadena de Texto");
```

o

```
String miCadena = new String("Mi Cadena de Texto");
System.out.println(miCadena);
```

ENLACE DE INTERÉS

Uno de los puntos más difíciles de la programación Java es la gestión de cadenas. Por esto se deja un enlace donde se podrá obtener más información sobre las operaciones con cadenas en Java.

- Gestión de cadenas

4.3 Funciones básicas con cadenas

Se van a ver los métodos que permiten manipular una cadena de texto:

- **Información básica de la cadena:**
  - **`length()`**- Devuelve el tamaño que tiene la cadena.
  - **`char charAt(int index)`**- Devuelve el carácter indicado como índice. El primer carácter de la cadena será el del índice 0. Junto con el método **`length()`** se pueden recuperar todos los caracteres de la cadena de texto.

Hay que tener cuidado. Ya que si se intenta acceder a un índice de un carácter que no existe nos devolverá una excepción del tipo `IndexOutOfBoundsException`.

- **Comparación de Cadenas:** Los métodos de comparación sirven para comparar si dos cadenas de texto son iguales o no. Dentro de los métodos de comparación están los siguientes:
  - **`boolean equals(Object anObject)`** - Permite comparar si dos cadenas de texto son iguales. En el caso de que sean iguales devolverá como valor "true", y en caso contrario devolverá "false". Este método tiene en cuenta si los caracteres van en mayúsculas o en minúsculas. Si se quiere omitir esta validación se tienen dos opciones. La primera de las opciones es convertir las cadenas a mayúsculas o minúsculas con los métodos **`toUpperCase()`** y **`toLowerCase()`** respectivamente. La segunda opción es utilizar el método **`equalsIgnoreCase()`** que omite si el carácter está en mayúsculas o en minúsculas.
  - **`boolean equalsIgnoreCase(String anotherString)`** - Compara dos cadenas de caracteres omitiendo si los caracteres están en mayúsculas o en minúsculas.
  - **`int compareToIgnoreCase(String str)`** - Este método se comportará igual que el anterior, pero ignorando las mayúsculas. Todo un alivio por si se nos escapa algún carácter en mayúsculas.
  - **`int compareTo(String anotherString)`** - Este método es un poco más avanzado que el anterior, el cual, solo indicaba si las cadenas eran iguales o diferentes. En este caso se comparan las cadenas léxicamente. Para ello se basa en el valor Unicode de los caracteres. Se devuelve un **entero menor de 0** si la cadena sobre la que se parte es léxicamente **menor** que la cadena pasada como argumento. Si las dos cadenas son **iguales** léxicamente se devuelve un **0**. Si la cadena es **mayor** que la pasada como argumento se devuelve un **número entero positivo**. Pero qué significa "mayor, menor o igual léxicamente".
    Para describirlo se verá con un pequeño ejemplo en Java:

```
String s1 = "Cuervo";
String s2 = "Cuenca";
s1.compareTo(s2);
```

    Se comparan las dos cadenas. Los tres primeros caracteres son iguales "Cue".

Cuando el método llega al carácter de índice 3 (4º carácter) tiene que comparar entre la r minúscula y la n minúscula. Si utiliza el código Unicode llegará a la siguiente conclusión.

r(114) > n(110)

Y devolverá la resta de sus valores. En este caso un 4.

Se debe tener cuidado, porque este método no tiene en cuenta las mayúsculas y minúsculas. Y dichos caracteres, aun siendo iguales, tienen diferentes códigos. En la siguiente comparación:

```
String s1 = "CueRvo";
String s2 = "Cuervo";
s1.compareTo(s2);
```

Nuevamente los tres caracteres iniciales son iguales. Pero el cuarto es distinto. Por un lado, se tiene la r minúscula y por otro la r mayúscula. Así:

R(82) < r(114)

- **Búsqueda de caracteres:** Se tiene un conjunto de métodos que permiten buscar caracteres dentro de cadenas de texto. Y es que no ha de olvidar que la cadena de caracteres no es más que eso: una suma de caracteres.
  - **`int indexOf(int ch)`** - Devuelve la posición de un carácter dentro de la cadena de texto. En el caso de que el carácter buscado no exista devolverá -1. Si lo encuentra devuelve un número entero con la posición que ocupa en la cadena.
  - **`int indexOf(int ch, int fromIndex)`** - Realiza la misma operación que el anterior método, pero en vez de hacerlo a lo largo de toda la cadena lo hace desde el índice (fromIndex) que se le indique.
  - **`int lastIndexOf(int ch)`** - Indica cuál es la última posición que ocupa un carácter dentro de una cadena. Si el carácter no está en la cadena devuelve -1.
  - **`int lastIndexOf(int ch, int fromIndex)`** - Lo mismo que el anterior, pero a partir de una posición indicada como argumento.
- **Búsqueda de subcadenas:** Este conjunto de métodos es, probablemente, el más utilizado para el manejo de cadenas de caracteres. Van a permitir buscar cadenas dentro de cadenas, así como conocer la posición en la que se encuentran en la cadena origen para poder acceder a la subcadena. Dentro de este conjunto se encuentran:
  - **`int indexOf(String str)`** - Busca una cadena dentro de la cadena origen. Devuelve un entero con el índice a partir del cual está la cadena localizada. Si no encuentra la cadena devuelve un -1.

- **`int indexOf(String str, int fromIndex)`** - Misma funcionalidad que `indexOf(String str)`, pero a partir de un índice indicado como argumento del método.
- **`int lastIndexOf(String str)`** - Si la cadena que se busca se repite varias veces en la cadena origen, se puede utilizar este método que indicará el índice donde empieza la última repetición de la cadena buscada.
- **`lastIndexOf(String str, int fromIndex)`** - Lo mismo que el anterior, pero a partir de un índice pasado como argumento.
- **`boolean startsWith(String prefix)`** - Probablemente se haya encontrado con el problema de saber si una cadena de texto empieza con un texto específico. La verdad es que este método se podía obviar y utilizar el método `indexOf()`, con el cual, en el caso de que devolviese un 0, se sabe que es el inicio de la cadena.
- **`boolean startsWith(String prefix, int toffset)`** - Más elaborado que el anterior, y quizás, con un poco menos de significado que el anterior.
- **`boolean endsWith(String suffix)`** – Para resolver el problema de conocer si una cadena de texto acaba con otra. De igual manera que sucedía con el método `startsWith()` se puede utilizar una mezcla entre los métodos `indexOf()` y `length()` para reproducir el comportamiento de `endsWith()`.

ENLACE DE INTERÉS

En el siguiente enlace se puede ver un tutorial sobre la gestión de cadenas en C, que es otro lenguaje muy utilizado.

- Programación en C/Cadenas de caracteres

- **Métodos con subcadenas:** Ahora que se sabe cómo localizar una cadena dentro de otra se va a ver cómo se substrae de donde está.
  - **`String substring(int beginIndex)`** - Este método devolverá la cadena que se encuentra entre el índice pasado como argumento (`beginIndex`) y el final de la cadena origen. Así, si se tiene la siguiente cadena:

```
String s = "Víctor Cuervo";
```

El método…

```
s.substring(7);
```

Devolverá "Cuervo".

- **`String substring(int beginIndex, int endIndex)`** - Si se da el caso de que la cadena que se desea recuperar no llega hasta el final de la cadena origen, que será lo normal, se puede utilizar este método indicando el índice inicial y final del cual queremos obtener la cadena. Se debe tener en cuenta que el valor proporcionado a `endIndex` se interpretará como `endIndex-1`.

  Así, si se parte de la cadena...

  ```
  String s = "En un lugar de la mancha....";
  ```

  El método...

  ```
  s.substring(6,11);
  ```

  Devolverá la palabra "lugar", ya que tiene en cuenta desde el carácter que se encuentra en la posición 6 hasta el carácter de la posición 10 (=11-1).

5. TRATAMIENTO DE XML EN JAVA (LECTURA Y ESCRITURA)

En este apartado se explicará cómo procesar ficheros en formato XML, las API más importantes que se usan en Java y las diferencias básicas entre ellas. Estas dos API van a ser **DOM** y **SAX**.

Los ficheros XML se usan básicamente para tratar datos, ya sea para estructurarlos, para enviar y recibir datos o para utilizarlos como base de datos. La principal idea de los ficheros XML es que son portables, e independientes del lenguaje de programación que usemos para procesarlos, además de ser simples de editar a mano y fáciles de comprender.

Existen dos formas de procesar los ficheros XML en Java, básicamente. Por una parte, se tiene el modelo **DOM** (Document Object Model) y por otra parte el modelo **SAX** (Simple API for XML).

ENLACE DE INTERÉS

Si se necesita ampliar información sobre las librerías para el tratamiento de XML en Java se puede visitar:

- Librerías para el tratamiento de XML en Java

5.1 Dom

A la hora de procesar un documento XML con DOM, la representación que se tiene va a ser la de un árbol jerárquico en memoria. Esto implica varias cosas que se detallan a continuación:

- Se puede leer **cualquier parte del árbol** (todo es un nodo), de forma que se puede procesar de arriba a abajo pero también se puede volver atrás.
- Se puede **modificar cualquier nodo** del árbol.
- Al estar cargado en **memoria**, se puede tener una **falta** de ésta. Con ficheros XML pequeños no se tienen problemas, pero si el árbol es muy grande entonces se tendrá una falta de `'heap space'`.

Comentadas las características de DOM, se va a explicar cómo se van a procesar documentos con DOM. El árbol en memoria va a ser un `Document`. Para crear un objeto de esta clase se usarán las factorías de `Document`, de la siguiente manera:

```
try {
	DocumentBuilderFactory dbf =
				DocumentBuilderFactory.newInstance();
	DocumentBuilder db = dbf.newDocumentBuilder();
	Document doc = db.newDocument();
} catch (ParserConfigurationException e) {
	e.printStackTrace();
}


// Forma compacta
try {
	Document doc = DocumentBuilderFactory.newInstance()
				.newDocumentBuilder().newDocument();
} catch (ParserConfigurationException e) {
	e.printStackTrace();
}
```

Se tendrá un `Document` en memoria que representa al árbol del que saldrá el fichero XML. Sin embargo, este árbol no tiene ni siquiera un nodo raíz, así que el siguiente paso es crearlo:

```
Element root = doc.createElement("Raiz");
```

Se da por hecho que la variable `doc` ya existe porque se ha creado en el paso previo. El parámetro `String` que recibe la función `createElement()` es el texto de la etiqueta. Cada vez que se quiera crear un nuevo nodo, se deberá llamar a esta función.

En caso de querer añadir el texto correspondiente a un nodo, se usará la función `createTextElement()`, que recibe un `String` que será el texto que contenga.

```
Element nodo = doc.createElement("NombreElemento");
Text texto = doc.createTextNode("Texto del elemento");
```

Existe una cosa más en los ficheros XML, y son los **atributos**. Un **nodo** (una etiqueta), puede tener una serie de atributos a los cuales se les asigna nombre y valor, o puede no tener ninguno. De esta forma se podría agregar al nodo raíz el atributo `autor` con valor `elAutor`, de la siguiente forma:

```
root.setAttribute("autor", "yoAutor");
```

En cualquiera de los casos, en DOM **todo es un nodo**, de modo que la forma de agregar nodos es la misma, independientemente del tipo de nodo que sea. Sabiendo esto, solamente faltaría agregar cada nodo a su nodo padre:

```
nodo.appendChild(texto);
root.appendChild(nodo);
doc.appendChild(root);
```

Con esto hemos conseguido tener un `Document` creado y cargado en memoria. Sin embargo, habría que darle formato para posteriormente escribirlo en algún lugar. En este caso se escribe en un fichero de texto:

```
// Meta @ elAutor
	try {
		// Se vuelca el XML al fichero
		TransformerFactory transFact =
					TransformerFactory.newInstance();
		// Se añade el sangrado y la cabecera de XML
		transFact.setAttribute("indent-number", new Integer(3));
		Transformer trans = transFact.newTransformer();
		trans.setOutputProperty(OutputKeys.INDENT, "yes");
		trans.setOutputProperty(OutputKeys.OMIT_XML_DECLARATION,
"no");
		// Se hace la transformación
		StringWriter sw = new StringWriter();
		StreamResult sr = new StreamResult(sw);
		DOMSource domSource = new DOMSource(dom);
		trans.transform(domSource, sr);
	} catch(Exception ex) {
		ex.printStackTrace();
	}
```

Esto deja en la variable `sw` la representación XML de este documento, con su sangrado correspondiente y listo para ser tratado. Ahora se podría escribir por pantalla, por fichero, mandarlo por un Socket… en este caso se agrega un poco de código para escribirlo en un fichero:

```
// Meta @ elAutor
try {
	// Se crea el fichero para escribir en modo texto
	PrintWriter writer = new PrintWriter(new
					FileWriter("test.xml"));
	// Se escribe todo el árbol en el fichero
	writer.println(sw.toString());
	// Se cierra el fichero
	writer.close();
} catch (IOException e) {
	e.printStackTrace();
}
```

Teniendo esto se habrá conseguido un fichero de texto XML a partir de un árbol cuya representación será de la siguiente forma:

```
<Raiz autor="elAutor">
	<NombreElemento>Texto del elemento</NombreElemento>
</Raiz>
```

Por otra parte, si se quiere leer un fichero XML y procesarlo de alguna manera, primero se necesita crear un `Document` en memoria a partir de un fichero XML bien formado:

```
// Meta @ elAutor
try {
	DocumentBuilderFactory dbf =
				DocumentBuilderFactory.newInstance();
	DocumentBuilder db = dbf.newDocumentBuilder();
	Document doc = db.parse(new File("test.xml"));
	doc.getDocumentElement().normalize();
} catch (ParserConfigurationException e) {
	e.printStackTrace();
} catch (SAXException e) {
	e.printStackTrace();
} catch (IOException e) {
	e.printStackTrace();
}
```

Es importante tener en cuenta que se puede crear el `Document` a partir de una sola línea de código, con todos los constructores anidados.

Si se tiene un fichero XML mal formado se recibirá una excepción. La función `normalize()` **elimina** nodos de texto vacíos y combina los adyacentes (en caso de que los hubiera).

Para poder acceder al nodo raíz del documento, se va a utilizar la función **`getDocumentElement()`** del `Document`. Y a partir de aquí se puede empezar a recorrer el árbol. Cuando se tiene un nodo, por ejemplo, el nodo raíz, se pueden obtener todos los nodos que cuelgan de él con la función **`getChildNodes()`**. Sin embargo, esta función puede dar lugar a errores, y se va a ver en un ejemplo.

Se tiene el fichero generado en la parte anterior, y se lee con el código que se acaba de poner. Si se pidieran los hijos de nuestro nodo raíz, se esperaría obtener sólo un nodo, que en este caso será el nodo con nombre `NombreElemento`. Pero lo obtenido es diferente:

```
System.out.println(doc.getDocumentElement().getChildNodes().getLength(
));
// salida: 3
```

Se ve qué nodos son estos:

```
NodeList nodosRaiz = doc.getDocumentElement().getChildNodes();
for(int i = 0; i < nodosRaiz.getLength(); i++) {
	System.out.println(nodosRaiz.item(i).getNodeName());
}
```

Además, si se miran esos nodos de tipo `#text` se verá que no son nada. De modo que para evitarlos se pueden filtrar así:

```
if(!nodosRaiz.item(i).getNodeName().equals("#text")…
```

O también se podrían seleccionar los nodos que se deseen por el nombre, puesto que se sabe el nombre de los nodos de cada nivel:

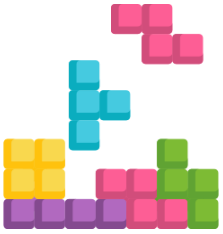
```
NodeList nodosRaiz =
	doc.getDocumentElement().getElementsByTagName("NombreElemento");
System.out.println(nodosRaiz.getLength());
System.out.println(nodosRaiz.item(0).getNodeName());
```

Es decir, la manera de ir leyendo los diferentes hijos de cada nodo es usando o bien **`getChildNodes()`** e ir filtrando por nombres (con un `case`, por ejemplo), o bien usar varios **`getElementsByTagName()`** en caso de tener etiquetas de nombres diferentes en el mismo nivel. Hay que recordar que el nodo raíz tenía un atributo.

Se supone que se quiere sacar el valor por alguna razón. El código sería el siguiente:

```
System.out.println(doc.getDocumentElement().getAttribute("autor"));
```

En caso de no ser el nodo raíz no se pondría **`doc.getDocumentElement()`** sino el `Element` correspondiente.



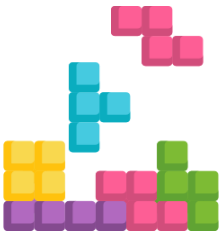
EJEMPLO PRÁCTICO

Se ve un ejemplo completo.

Archivo XML:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
```

```
<Pedidos>
   <Pedido numeroPedido="1">
      <Cliente>
         <Nombre>Thomas</Nombre>
         <Apellido>Anderson</Apellido>
         <User>Neo</User>
      </Cliente>
      <Productos>
         <Producto>
            <Nombre>Matrix 7.0</Nombre>
            <FechaAlta>1307542390468</FechaAlta>
         </Producto>
         <Producto>
            <Nombre>Gravedad 2.0</Nombre>
<FechaAlta>1307542390468</FechaAlta>
         </Producto>
      </Productos>
   </Pedido>
   <Pedido numeroPedido="2">
      <Cliente>
         <Nombre>Mr.</Nombre>
         <Apellido>Regan</Apellido>
         <User>Cifra</User>
      </Cliente>
           <FechaAlta>1307542390468</FechaAlta>
        </Producto>
     </Productos>
  </Pedido>
  <Pedido numeroPedido="2">
     <Cliente>
        <Nombre>Mr.</Nombre>
        <Apellido>Regan</Apellido>
        <User>Cifra</User>
     </Cliente>
     <Productos>
        <Producto>
           <Nombre>Reinsercion 1.0</Nombre>
           <FechaAlta>1307542390468</FechaAlta>
        </Producto>
     </Productos>
  </Pedido>
</Pedidos>
```



EJEMPLO PRÁCTICO

Listado Clase Java

```
import java.io.File;
import java.io.FileWriter;
import java.io.IOException;
import java.io.PrintWriter;
import java.io.StringWriter;
import java.util.ArrayList;
import java.util.Date;

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.TransformerFactory;
import javax.xml.transform.dom.DOMSource;
import javax.xml.transform.stream.StreamResult;

import org.w3c.dom.Document;
import org.w3c.dom.Element;
import org.w3c.dom.Node;
import org.w3c.dom.NodeList;
import org.w3c.dom.Text;
import org.xml.sax.SAXException;

public class PedidosXML_DOM {

	private Document dom;

	public PedidosXML_DOM() {
		dom = null;
	}
	private void addCliente(Usuario cliente, Element nodoCliente) {
		Element nombre = dom.createElement("Nombre");
		Text textoNombre =
dom.createTextNode(cliente.getNombre());
		nombre.appendChild(textoNombre);
```

```
                Element apellido = dom.createElement("Apellido");
                Text textoApellido =
dom.createTextNode(cliente.getApellido());
                apellido.appendChild(textoApellido);

                Element user = dom.createElement("User");
                Text textoUser = dom.createTextNode(cliente.getUser());
                user.appendChild(textoUser);

                nodoCliente.appendChild(nombre);
                nodoCliente.appendChild(apellido);
                nodoCliente.appendChild(user);
        }
        private void addProducto(Producto producto, Element nodoProducto) {
                Element nombre = dom.createElement("Nombre");
                Text textoNombre =
dom.createTextNode(producto.getNombre());
                nombre.appendChild(textoNombre);

                Element fechaAlta = dom.createElement("FechaAlta");
                Text textoFechaAlta =
                        dom.createTextNode(String.
valueOf(producto.getFechaAlta().getTime()));
                fechaAlta.appendChild(textoFechaAlta);

                nodoProducto.appendChild(nombre);
                nodoProducto.appendChild(fechaAlta);
        }

         private void addProductos(ArrayList<Producto> listaProductos,
Element nodoListaProductos) {
                for(int i = 0; i < listaProductos.size(); i++) {
                        Element producto = dom.createElement("Producto");
                        addProducto(listaProductos.get(i), producto);
                        nodoListaProductos.appendChild(producto);
                }
        }
        private void addPedido(Pedido pedido) {
                // seleccionamos la raiz
                Element root = dom.getDocumentElement();

                // Se crea un nuevo elemento con el atributo del
                // número de producto
                Element unPedido = dom.createElement("Pedido");
                unPedido.setAttribute("numeroPedido",
                                String.valueOf(pedido.getNumeroPedido()));
```

```
                // Se crea un nuevo elemento para el cliente
                Element cliente = dom.createElement("Cliente");
                addCliente(pedido.getCliente(), cliente);

                // Se crea un nuevo elemento para los productos
                // de los que consta el pedido
                Element productos = dom.createElement("Productos");
                addProductos(pedido.getListaProductos(), productos);

                // Se inserta el cliente y los productos
                // en el elemento del pedido
                unPedido.appendChild(cliente);
                unPedido.appendChild(productos);

                // Se inserta el pedido en la raíz
                root.appendChild(unPedido);
                /*
                 * Escribe en el fichero la representación del árbol XML
                 */
                private void write(StringWriter sw, String path) {
                        try {
                                // Se crea un fichero para escribir en modo
texto
                                PrintWriter writer = new PrintWriter(
                                                new
FileWriter(path));
                                // Se escribe todo el árbol en XML
                                writer.println(sw.toString());
                                // Se cierra el fichero
                                writer.close();
                        } catch (IOException e) {
                                e.printStackTrace();
                        }
                }

                /*
                   * Transforma el árbol, agregando la cabecera y añadiendo
sangrados
                 */
                private void toFile(String ruta) {
                        try {
                                // Se vuelca el XML al fichero
                                TransformerFactory transFact =
TransformerFactory.newInstance();

                                // Se añade el sangrado
                                transFact.setAttribute("indent-number",
                                                new
Integer(3));
```

```
                                Transformer trans =
transFact.newTransformer();
                            // Se incluye la cabecera XML y el sangrado
                            trans.setOutputProperty(

OutputKeys.OMIT_XML_DECLARATION, "no");
                                trans.setOutputProperty(OutputKeys.INDENT,
"yes");

                            // Se hace la transformación
                            StringWriter sw = new StringWriter();
                            StreamResult sr = new StreamResult(sw);
                            DOMSource domSource = new DOMSource(dom);
                            trans.transform(domSource, sr);

                            // Se escribe en el fichero
                            write(sw, ruta);
                        } catch(Exception ex) {
                            ex.printStackTrace();
                        }
                }

                /*
                 * Se crea un fichero XML a partir de una lista de pedidos
                 */

public void pedidosToXML(ArrayList<Pedido> pedidos, String ruta) {
                        // Se crea un nuevo Document donde se va a guardar
                        // toda la estructura
                        try {
                            dom = DocumentBuilderFactory.newInstance().

newDocumentBuilder().newDocument();
                            Element root = dom.createElement("Pedidos");
                            dom.appendChild(root);
                        } catch (ParserConfigurationException e) {
                            e.printStackTrace();
                        }

                        for(int i = 0; i < pedidos.size(); i++)
                            addPedido(pedidos.get(i));

                        // Se vuelca por pantalla
                        toFile(ruta);
                }

                // Lectura de XML con DOM
                private Usuario readUsuario(Node nodoUsuario) {
```

```
                        Element elementoUsuario = (Element)nodoUsuario;
                        String nombre = elementoUsuario.

getElementsByTagName("Nombre").item(0).getTextContent();
                        String apellido = elementoUsuario.

getElementsByTagName("Apellido").item(0).getTextContent();
                        String user = elementoUsuario.

getElementsByTagName("User").item(0).getTextContent();

                        return new Usuario(nombre, apellido, user);
                }

                private Producto readProducto(Node nodoProducto) {
                        Element elementoProducto = (Element)nodoProducto;

                        String nombre = elementoProducto.

getElementsByTagName("Nombre").item(0).getTextContent();
                        long fechaAlta = Long.valueOf(elementoProducto.

getElementsByTagName("FechaAlta").item(0).getTextContent());

                        return new Producto(nombre, new Date(fechaAlta));
                                }

                private Pedido readPedido(Node nodoPedido) {
                        Element elementoPedido = (Element)nodoPedido;

                        int numeroPedido = Integer.valueOf(

elementoPedido.getAttribute("numeroPedido"));
                        Usuario user = readUsuario(elementoPedido.

getElementsByTagName("Cliente").item(0));
                        Pedido pedido = new Pedido(numeroPedido, user);

                        NodeList productos = elementoPedido.

getElementsByTagName("Productos");
                        for(int i = 0; i < productos.getLength(); i++) {
                                pedido.addProducto(readProducto

((Element)productos.item(0)));
                        }
                        return pedido;
                }
```

```
		/*
		 * Se crea una lista de pedidos procesando un fichero XML
		 */
		public ArrayList<Pedido> XMLtoPedidos(String ruta) {
			ArrayList<Pedido> pedidos = new ArrayList<Pedido>();

			try {
				DocumentBuilderFactory dbf =

DocumentBuilderFactory.newInstance();
				 DocumentBuilder db =
dbf.newDocumentBuilder();
				dom = db.parse(new File(ruta));
				dom.getDocumentElement().normalize();

				 // Se seleccionan todos los pedidos y se
van leyendo
				 NodeList listaPedidos =
dom.getDocumentElement().

getElementsByTagName("Pedido");
				 for(int i = 0; i <
listaPedidos.getLength(); i++) {

pedidos.add(readPedido(listaPedidos.item(i)));
				}
			} catch (SAXException e) {
				e.printStackTrace();
			} catch (IOException e) {
				e.printStackTrace();
			} catch (ParserConfigurationException e) {
				e.printStackTrace();
			}

			return pedidos;
		}
	}
```

5.2 SAX

Al contrario que con DOM, al procesar en SAX **no se va a tener la representación completa del árbol en memoria**, pues SAX funciona con **eventos**. Esto implica:

- Al no tener el árbol completo **no se puede volver atrás**, pues se va leyendo secuencialmente.

- La **modificación** de un nodo es mucho **más compleja** (y la **inserción** de nuevos nodos).
- Como no se tiene el árbol en memoria es mucho más **`memory friendly`**, de modo que es la **única opción viable** para casos de **ficheros muy grandes**, pero demasiado complejo para ficheros pequeños.
- Al ser orientado a eventos, el **procesado** se vuelve **bastante complejo**.

De esta forma, no se va a explicar cómo escribir o modificar ficheros con SAX debido a su complejidad, sino cómo leerlos y procesarlos.

Se va a partir de que se tiene ya el fichero XML anterior:

```
<Raiz autor=" ">
      <NombreElemento>Texto del elemento</NombreElemento>
</Raiz>
```

Para poder procesar un fichero XML con SAX la clase lectora va a necesitar heredar de la clase `DefaultHandler` (se recuerda que, como buena práctica de programación, los datos deben estar separados de la entrada/salida). Además, se va a necesitar un objeto de la clase `XMLReader`, el cual va a usar la propia clase como `ContentHandler` y `ErrorHandler`. El esqueleto de la clase entonces sería algo así:

```
public class LectorXML_SAX extends DefaultHandler {
      private XMLReader reader;
      public LectorXML_SAX() {
            try {
                  reader = XMLReaderFactory.createXMLReader();
                  reader.setContentHandler(this);
                  reader.setErrorHandler(this);
            } catch (SAXException e) {
                  e.printStackTrace();
            }
      }
}
```

Se ha dicho que SAX funciona por eventos. Pero ¿qué eventos son esos?

- **`startDocument()`**: llamado cuando empieza el documento.
- **`endDocument()`**: llamado cuando acaba el documento.
- **`startElement()`**: llamado cuando empieza un nodo (por ejemplo, al llegar al < en <Raiz>).

- **`characters()`**: llamado al acabar el evento **`startElement()`**. Sirve para leer el contenido de una etiqueta (por ejemplo, el texto Texto del elemento de la etiqueta `NombreElemento`).
- **`endElement()`**: llamado al llegar al final de una etiqueta (por ejemplo, al llegar al </ en `</NombreElemento>`).

Ahora se creará el método para la lectura, de la siguiente manera:

```
public void leeXML(String path) {
	try {
		reader.parse(path);
	} catch (IOException e) {
		e.printStackTrace();
	} catch (SAXException e) {
		e.printStackTrace();
	}
}
```

Al ejecutar esto se puede observar que no se obtiene nada. Esto es así porque las funciones explicadas más arriba para los eventos están vacías y se necesita redefinirlas.

En este caso se va a volcar el contenido del fichero por pantalla, sin sangrado:

```
	@Override
	public void startElement(String uri, String localName, String name,
Attributes atts) {
		System.out.println("<" + localName + ">");
	}

	@Override
	public void characters(char[] cadena, int inicio, int length) {
	    if(String.valueOf(cadena, inicio, length).trim().length() != 0)
		System.out.println(String.valueOf(cadena, inicio, length));
	}

	@Override
	public void endElement(String uri, String name, String qName) {
		System.out.println("");
	}
```

Si se crease un objeto de esta clase y se invocase al método **`leerXML()`**, se obtendría lo siguiente por la consola:

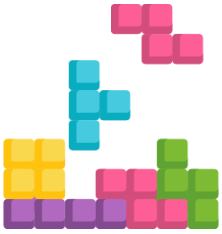
```
<Raiz><NombreElemento>Texto del elemento</NombreElemento></Raiz>
```

A partir de esto se podría, en lugar de escribir por pantalla los valores, procesarlos y crear atributos de objetos de clases que se hayan definido.

ARTÍCULO DE INTERÉS

En este artículo se puede ver una comparativa de las tecnologías para XML de Java:

- Comparación de las Tecnologías Java para XML



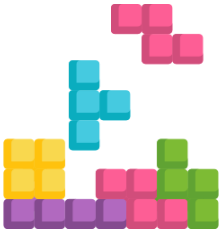
EJEMPLO PRÁCTICO

Se ve un ejemplo completo.

Archivo XML:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
```

```
<Pedidos>
  <Pedido numeroPedido="1">
     <Cliente>
        <Nombre>Thomas</Nombre>
        <Apellido>Anderson</Apellido>
        <User>Neo</User>
     </Cliente>
     <Productos>
        <Producto>
           <Nombre>Matrix 7.0</Nombre>
           <FechaAlta>1307542390468</FechaAlta>
        </Producto>
        <Producto>
           <Nombre>Gravedad 2.0</Nombre>
           <FechaAlta>1307542390468</FechaAlta>
        </Producto>
</Productos>
  </Pedido>
  <Pedido numeroPedido="2">
     <Cliente>
        <Nombre>Mr.</Nombre>
        <Apellido>Regan</Apellido>
        <User>Cifra</User>
     </Cliente>
    <Productos>
       <Producto>
          <Nombre>Reinsercion 1.0</Nombre>
          <FechaAlta>1307542390468</FechaAlta>
       </Producto>
    </Productos>
 </Pedido>
</Pedidos>
```



EJEMPLO PRÁCTICO

Listado de la clase Java

```
import java.io.IOException;
import java.util.ArrayList;

import org.xml.sax.Attributes;
import org.xml.sax.SAXException;
```

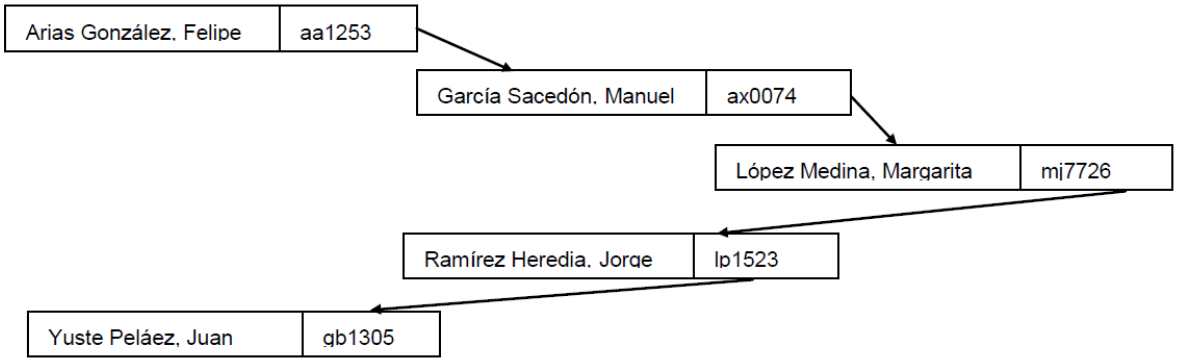
```
import org.xml.sax.XMLReader;
import org.xml.sax.helpers.DefaultHandler;

public class PedidosXML_SAX extends DefaultHandler {

	private XMLReader reader;

	public PedidosXML_SAX() {
		try {
			reader = XMLReaderFactory.createXMLReader();
			reader.setContentHandler(this);
			reader.setErrorHandler(this);
		} catch (SAXException e) {
			e.printStackTrace();
		}
	}
	public ArrayList<Pedido> XMLtoPedidos(String ruta) {
		ArrayList<Pedido> pedidos = new ArrayList<Pedido>();
		try {
			reader.parse(ruta);
		} catch (IOException e) {
			e.printStackTrace();
		} catch (SAXException e) {
			e.printStackTrace();
		}
		return pedidos;
	}

	@Override
	public void startElement(String uri, String localName, String name,
                                                        Attributes
atts) {
		System.out.println("<" + localName + ">");
	}

	@Override
	public void characters(char[] cadena, int inicio, int length) {
		if(String.valueOf(cadena, inicio, length).trim().length() !=
0)
			System.out.println(String.valueOf(cadena, inicio,
length));
	}

	@Override
	public void endElement(String uri, String name, String qName) {
		System.out.println("</" + name +">");
	}
}
```

6. LISTAS

Una lista es una estructura de datos lineal que se puede representar simbólicamente como un conjunto de nodos enlazados entre sí.

Las listas permiten modelar diversas entidades del mundo real como, por ejemplo, los datos de los alumnos de un grupo académico, los datos del personal de una empresa, los programas informáticos almacenados en un disco magnético, etc.

La figura muestra un ejemplo de lista correspondiente a los nombres y apellidos de un conjunto de alumnos con su código de matrícula.



Tal vez resulte conveniente identificar a los diferentes elementos de la lista (que normalmente estarán configurados como una estructura de registro) mediante uno de sus campos (clave) y en su caso, se almacenará la lista respetando un criterio de ordenación (ascendente o descendente) respecto al campo clave.

ENLACE DE INTERÉS

Para ampliar información sobre las listas y su tratamiento en Java visite el siguiente enlace:

- Manejo de Listas [Java]

Una **definición formal** de lista es la siguiente: *"Una lista es una secuencia de elementos del mismo tipo, de cada uno de los cuales se puede decir cuál es su siguiente (en caso de existir)."*

Existen dos criterios generales de calificación de listas:

- **Por la forma de acceder** a sus elementos.

- **Listas densas**. Cuando la estructura que contiene la lista es la que determina la posición del siguiente elemento. La localización de un elemento de la lista es la siguiente:
  - Está en la posición 1 si no existe elemento anterior.
  - Está en la posición N si la localización del elemento anterior es la posición (N-1).
- **Listas enlazadas**: La localización de un elemento es:
  - Estará en la dirección k, si es el primer elemento, siendo k conocido.
  - Si no es el primer elemento de la lista, estará en una dirección, j, que está contenida en el elemento anterior.

- **Por la información** utilizada para acceder a sus elementos:
  - **Listas ordinales**. La posición de los elementos en la estructura la determina su orden de llegada.
  - **Listas calificadas**. Se accede a un elemento por un valor que coincide con el de un determinado campo, conocido como clave. Este tipo de listas se pueden clasificar a su vez en ordenadas o no ordenadas por el campo clave.

ENLACE DE INTERÉS

Antes de seguir con la unidad es recomendable que visite el siguiente enlace donde obtendrá más información sobre las estructuras de datos:

- Código de programación

6.1 Implementación de listas

El concepto de lista puede implementarse en soportes informáticos de diferentes maneras.

- **Mediante estructuras estáticas**. Con toda seguridad resulta el mecanismo más intuitivo. Una simple *matriz* resuelve la idea.

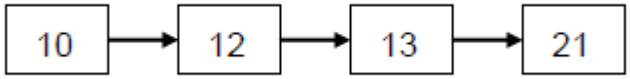
0	Arias González, Felipe	aa1253
1	García Sacedón, Manuel	ax0074
2	López Medina, Margarita	mj7726
3	Ramírez Heredia, Jorge	lp1523
4	Yuste Peláez, Juan	gb1305

Imagen: Implementación de una lista densa mediante una estructura estática (matriz)

El **problema** de esta alternativa es el derivado de las operaciones de **inserción** y **modificación**.
En efecto, la declaración de una lista mediante una matriz implica **conocer de antemano** el número (o al menos el orden de magnitud) de elementos que va a almacenar, pudiendo darse las circunstancias de que si se declara con un número pequeño podría desbordarse su capacidad o, en caso contrario, declararlo desproporcionadamente con un número elevado provocaría un decremento de eficiencia.
Otro problema asociado es el tratamiento de los **elementos eliminados**. Dado que, en el caso de no informar, de alguna manera, de la inexistencia de dicho elemento el nodo previamente ocupado (y ahora no válido) quedaría como no disponible.
Adicionalmente, si se desea trabajar con listas ordenadas el algoritmo de inserción debería alojar a los nuevos elementos en la posición o con la referencia adecuada.
Algunas soluciones, más o menos ingeniosas, permiten tratar estas circunstancias. La figura siguiente muestra un ejemplo basado en una matriz de registros.

0	1	2	3	4	5	6	7	8
1	10	77	12	26	21	11	13	18
2	3	4	7	6	0	8	5	0

Otra posible representación de esta lista sería la siguiente:

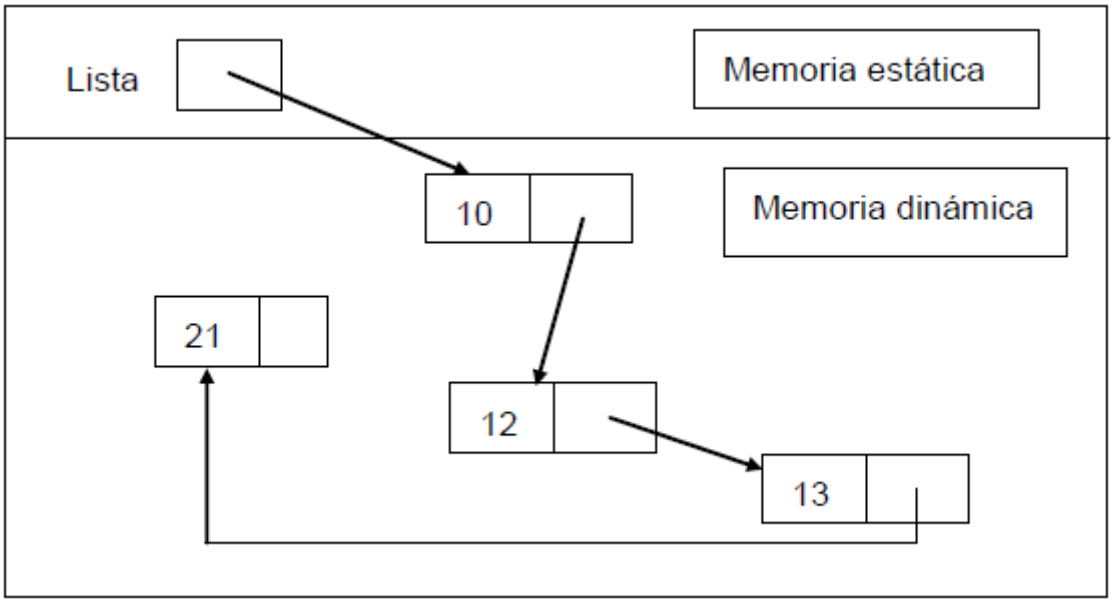


VIDEO DE INTERÉS

Funcionamiento de las listas enlazadas en Java:

- Tutoriales Java: 18 Listas Enlazadas (Punteros)

- **Mediante estructuras dinámicas.** Sin duda se trata de **la mejor alternativa** para la construcción de listas. Se trata de hacer uso de la correspondiente tecnología que implica el uso de referencias.
  La idea consiste en declarar una **referencia a un nodo** que será el primero de la lista. Cada nodo de la lista (en la memoria dinámica) contendrá tanto la propia información **del mismo** como una referencia al **nodo siguiente**. Se deberá establecer un convenio para identificar el **final de la lista**.
  La figura muestra un ejemplo de implementación dinámica de la lista de la figura 3.



Lo explicado hasta el momento consiste en la solución más sencilla: cada nodo de la lista "apunta" al siguiente, con las excepciones del **último elemento** de la lista (su "apuntador", o "**referencia**" es un valor especial, por ejemplo, *null*, en Java) y del **primero**, que es "apuntado" por la referencia declarada en la memoria estática. Esta tecnología se identifica como "Listas enlazadas o unidireccionales". Existen otras posibilidades entre las que cabe mencionar: listas bidireccionales o doblemente enlazadas, listas circulares, listas con cabecera, listas con centinela o cualquier combinación de ellas.

6.2 Tratamiento de listas en java

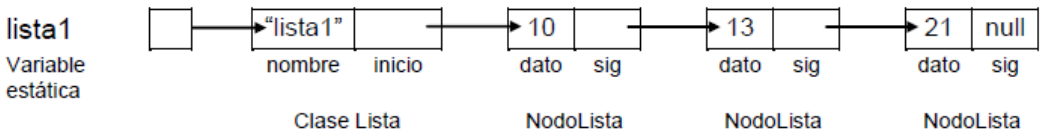
Para la utilización de listas es necesario definir la clase *NodoLista* utilizando la siguiente sintaxis:

```
public class NodoLista {
	public int dato;
	public NodoLista sig;

	public NodoLista(int x, NodoLista n) {
		dato = x;
		sig = n;
	}
}
```

Así como la clase Lista:

```
public class Lista {
	public Lista(String nombreLista) {
		inicio = null;
		nombre = nombreLista;
	}

	public NodoLista inicio;
	public String nombre;
}
```

La representación gráfica de una variable de la clase `Lista` llamada *lista1* que contenga los elementos 10, 13 y 21 sería:



6.3 Algoritmos básicos con listas

Los algoritmos que implican el recorrido, parcial o total, de la lista pueden implementarse tanto de forma recursiva como iterativa.

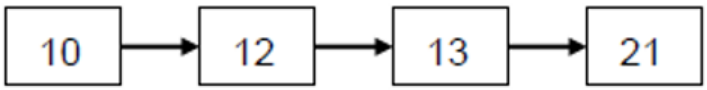
El recorrido de una lista de manera **recursiva** se puede realizar por medio de un método ***static*** invocado desde un programa que utilice la clase `Lista`. Esto implica utilizar el tipo `NodoLista` para avanzar por la lista y ver el contenido del campo clave. Merecen una consideración especial:

- La llamada inicial donde *lista* es el valor de la variable estática.
- El final del recorrido, que se alcanza cuando la lista está vacía.

En el siguiente método estático (`escribirLista`), se recorre una lista (`nodoLista`) mostrando en la pantalla el contenido de sus campos clave. Se utiliza un método de llamada (`escribirListaCompleta`), que recibe como argumento un objeto de la clase *Lista* (*lista*):

```
static void escribirLista(NodoLista nodoLista) {
	if (nodoLista != null) {
		System.out.print(nodoLista.clave + " ");
		escribirLista(nodoLista.sig);
	} else
		System.out.println(" FIN");
}

static void escribirListaCompleta(Lista lista) {
	if (lista != null) {
		System.out.print(lista.nombre + ": ");
		escribirLista(lista.inicio);
	} else
		System.out.println("Lista vacía");
}
```

Si se aplica el algoritmo anterior a la lista de la figura:



El resultado sería la secuencia siguiente:

lista1: 10 13 21 FIN.

La ejecución de *escribirListaCompleta* implicaría una llamada inicial al método *escribirLista,* pasando como argumento *lista.inicio* (la referencia al nodo de clave 10) y 3 llamadas recursivas al método *escribirLista*:

- En la primera llamada recursiva, *nodoLista* es el contenido del campo *sig* del nodo de clave 10. Es decir, una referencia al nodo de clave 13.
- En la segunda, *nodoLista* es el contenido del campo *sig* del nodo de clave 13. Es decir, una referencia al nodo de clave 21.

- En la tercera, *nodoLista* es el contenido del campo *sig* del nodo de clave 21, es decir, *null*. Cuando se ejecuta esta tercera llamada se cumple la condición de finalización y, en consecuencia, se inicia el proceso de "vuelta". Ahora *nodoLista* toma sucesivamente los valores:
  - Referencia al nodo de clave 21 (campo *sig* del nodo de clave 13).
  - Referencia al nodo de clave 13 (campo *sig* del nodo de clave 10).
  - Referencia al nodo de clave 10 (el primer elemento de la lista).

El **recorrido de una lista** es una operación necesaria en los algoritmos de **eliminación** y, en muchos casos, también en los de **inserción**. La condición de finalización "**pesimista**" consiste en alcanzar el final de la lista (*nodoLista == null*). No obstante, normalmente se produce una **terminación anticipada** que se implementa *no realizando nuevas llamadas recursivas*.

En los siguientes apartados se van a presentar los diferentes tipos de listas, sobre las cuales se explican algunos algoritmos básicos: *inicialización, búsqueda, inserción y eliminación.*

6.4 Listas ordinales

En las listas ordinales el **orden** dentro de la estructura lo establece la **llegada** a la misma. A diferencia de las listas calificadas, en este tipo de listas no existe ningún elemento que identifique el nodo, y, por lo tanto, los valores **se pueden repetir**. El criterio de inserción resulta específico en cada caso (se podría insertar por el principio, o bien por el final).

ENLACE DE INTERÉS

A continuación, se van a estudiar dos ejemplos de listas ordinales: las pilas y las colas. Antes de comenzar visite el siguiente enlace para entender mejor de en qué consisten:

- Java pilas (Stacks) y colas (Queues)

Veremos a continuación dos ejemplos de listas ordinales que ya hemos tratado como TADs (*Tipos Abstractos de Datos*): las pilas y las colas.

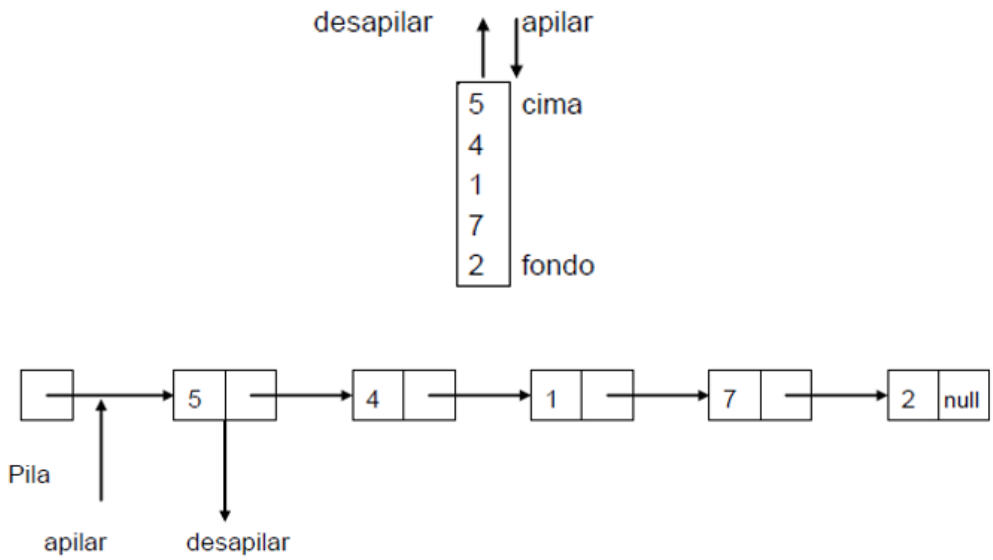
6.4.1 Pilas

Una pila es una agrupación de elementos de determinada naturaleza o tipo (datos de personas, números, procesos informáticos, automóviles, etc.) entre los que existe definida una relación de orden (**estructura de datos**).

En función del tiempo, algunos elementos de dicha naturaleza pueden llegar a la pila o salir de ella (**operaciones / acciones**). En consecuencia, el estado de la pila varía.

En una pila (comportamiento **LIFO** -*Last In First Out*-) se establece el criterio de ordenación en **sentido inverso al orden de llegada**. Así pues, el último elemento que llegó al conjunto será el primero en salir del mismo, y así sucesivamente.

Las figuras siguientes ilustran respectivamente el concepto de pila de números enteros y su implementación mediante una lista dinámica.



ENLACE DE INTERÉS

Una de las clases más importantes en Java para la implementación de Pilas es la clase `Stack`. Amplíe información visitando:

- La estructura de datos pila en java

La estructura de datos de la pila y el constructor utilizado sería:

```java
package tadPila;

//En esta clase se define el nodo:
class NodoPila {
	// Atributos accesibles desde otras rutinas del paquete
		int dato;
		NodoPila siguiente;

		// Constructor
	NodoPila(int elemento, NodoPila n) {
		dato = elemento;
		siguiente = n;
	}

}
```

Y la interfaz utilizada sería la siguiente:

```java
package tadPila;

import java.io.*;

public interface Pila {
	void inicializarPila();
	boolean pilaVacia();
	void eliminarPila();
	int cima() throws PilaVacia;
	void apilar(int x);
	int desapilar() throws PilaVacia;
	void decapitar() throws PilaVacia;
	void imprimirPila();
	void leerPila() throws NumberFormatException, IOException;
	int numElemPila();
}
```

A continuación, se muestra la clase `TadPila`, con el constructor y los algoritmos correspondientes a las operaciones *`pilaVacia`*, *`inicializarPila`*, así como *`apilar`* y *`desapilar`* que operan al principio de la lista por razones de eficiencia.

```
package tadPila;

public class TadPila implements Pila {
	protected NodoPila pila;

	public TadPila() {
		pila = null;
	}

	public void inicializarPila() {
		pila = null;
	}

	public boolean pilaVacia() {
		return pila == null;
	}

	public void apilar(int dato) {
		NodoPila aux = new NodoPila(dato, pila);
		pila = aux;
	}

	public int desapilar() throws PilaVacia {
		int resultado;
		if (pilaVacia())
			throw new PilaVacia("Desapilar: La pila está
					vacía");
		resultado = pila.dato;
		pila = pila.siguiente;
		return resultado;
	}
}
```

6.4.2 Colas

Una cola es una agrupación de elementos de determinada naturaleza o tipo (datos de personas, números, procesos informáticos, automóviles, etc.) entre los que existe definida una **relación de orden**. En función del tiempo, pueden llegar a la cola o salir de ella algunos elementos de dicha naturaleza (**operaciones/acciones**). En consecuencia, el estado de la cola varía.

En una cola (comportamiento **FIFO** -*First In First Out*-) se respeta como criterio de ordenación el momento de la llegada: el primero de la cola será el que primero llegó a ella y, en consecuencia, el primero que saldrá, y así sucesivamente.

Las figuras siguientes ilustran respectivamente el concepto de cola de números enteros y su implementación mediante una lista dinámica.

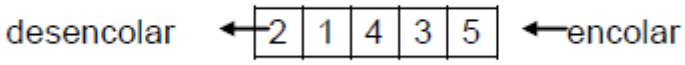
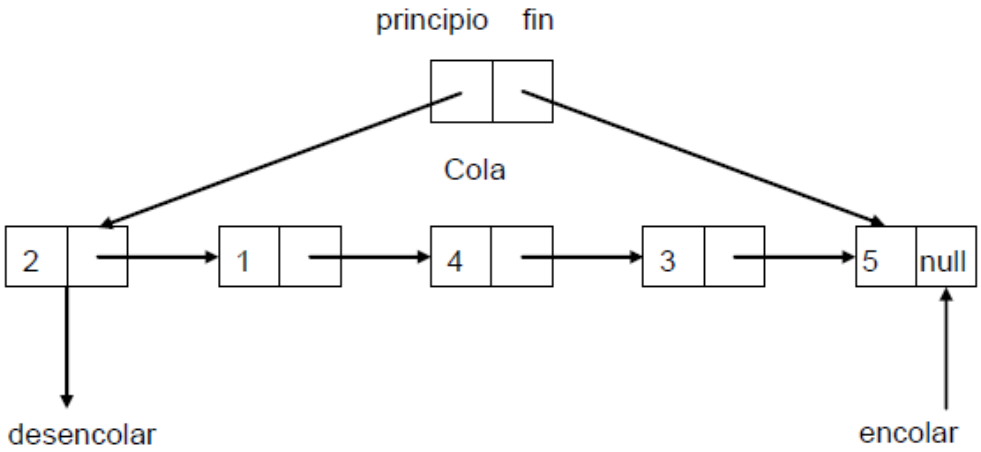


Imagen: Modelo gráfico de Cola



ENLACE DE INTERÉS

Amplíe información sobre esta estructura dinámica visitando:

- Estructuras dinámicas: Listas tipo Cola

La estructura de datos de la cola y el constructor sería:

```java
package tadCola;

//En esta clase se define el nodo:

class NodoCola {
	//Atributos accesibles desde otras rutinas del paquete
		int dato;
		NodoCola siguiente;
	//Constructor
	NodoCola(int elemento, NodoCola n) {
		dato = elemento;
		siguiente = n;
	}
}
```

Y la interfaz utilizada sería:

```java
package tadCola;

import java.io.IOException;

public interface Cola {
	void inicializarCola();
	boolean colaVacia();
	void eliminarCola();
	int primero() throws ColaVacia;
	void encolar(int x);
	int desencolar () throws ColaVacia;
	void quitarPrimero() throws ColaVacia;
	void mostrarEstadoCola();
	void imprimirCola();
	void leerCola() throws NumberFormatException, IOException;
	int numElemCola();
	void invertirCola() throws ColaVacia;
}
```

RESUMEN FINAL

Cualquier programa que se desarrolle en **Java** y que tenga la necesidad de manejar una cantidad importante de datos los organizará en arrays, listas o en archivos XML que se parsearán utilizando DOM o SAX. El uso de la clase String en Java es importante y no trivial: hay que tener en cuenta el funcionamiento de esta clase para trabajar de forma adecuada.

En esta unidad se ha comenzado analizando el concepto de array unidimensional y de más de una dimensión. Posteriormente se ha visto el funcionamiento de la clase String, para pasar a ver el funcionamiento de la interpretación de archivos XML utilizando SAX y DOM. Para finalizar esta unidad nos hemos adentrado en el funcionamiento de alguna estructura de datos dinámica como son las listas.