

4. Estructuras de datos: Arrays y matrices

- 1.- Introducción
- 2.- Arrays
 - 2.1.- Declaración y creación
 - 2.2.- Acceso a los componentes
 - 2.3.- Inicialización
 - 2.4.- Un ejemplo práctico
 - 2.5.- Arrays como parámetros. Paso de parámetros por referencia.
 - 2.6.- El atributo length
- 3.- Problemas de recorrido y búsqueda.
 - 3.1.- Problemas de recorrido
 - 3.2.- Problemas de búsqueda
- 3.- La clase Arrays.
- 4.- Listas : ArrayList
 - 4.1.- La clase ArrayList
- 4.- Arrays bidimensionales: matrices
 - 4.1.- Matrices en Java
 - 4.2.- Declaración de matrices.
 - 4.3.- Inicialización.
 - 4.3.- Recorrido
- 5.- Arrays multidimensionales

1.- Introducción

A menudo, para resolver problemas de programación, no basta con disponer de sentencias condicionales o iterativas como las que hemos visto (if, switch, while, for,...). También es necesario disponer de herramientas para organizar la información de forma adecuada: las estructuras de datos.

Los arrays son una estructura de datos fundamental, que está disponible en la mayoría de lenguajes de programación y que nos permitirá resolver problemas que, sin ellos, resultarían difíciles o tediosos de solucionar.

Imaginemos, por ejemplo, que queremos leer los datos de pluviosidad de cada uno de los 31 días de un mes. Posteriormente se desea mostrar la pluviosidad media del mes y en cuántos días las lluvias superaron la media.

Con las herramientas de que disponemos hasta ahora, nos veríamos obligados a declarar 31 variables *double*, una para cada día, y a elaborar un largo programa que leyera los datos y contara cuales superan la media. Con el uso de arrays, problemas como este tienen una solución fácil y corta.

2.- Arrays

Un array es una colección de elementos del **mismo tipo**, que tienen un **nombre o identificador común**.

Se puede acceder a cada componente del array de forma individual para consultar o modificar su valor. El acceso a los componentes se realiza mediante un subíndice, que viene dado por la posición que ocupa el elemento dentro del array.

En la siguiente figura se muestra un array *c* de enteros

c	-1	8	23	5	12	-5	255	-28	30	42
	c[0]	c[1]	c[2]	c[3]	c[4]	c[5]	c[6]	c[7]	c[8]	c[9]

El primer subíndice de un array es el cero. El último subíndice es la longitud del array menos uno.

El número de componentes de un array se establece inicialmente al crearlo y no es posible cambiarlo de tamaño. Es por esto que reciben el nombre de **estructuras de datos estáticas**.

2.1.- Declaración y creación

Para poder utilizar un array hay que **declararlo** y **crearlo**:

tipo nombreVariable[] = **new** tipo[numElementos];

o

tipo[] nombreVariable = **new** tipo[numElementos];

En la **declaración** se establece el **nombre** de la variable y el **tipo** de los componentes. Por ejemplo:

```
double lluvia[ ] ; // lluvia es un array de double  
double[] lluvia ; // lluvia es un array de double
```

En la declaración anterior no se ha establecido el número de componentes. El número de componentes se indica en la creación, que se hace utilizando el operador **new**:

```
lluvia = new double [31];
```

Con esta instrucción se establece que el número de elementos del array *lluvia* es 31, reservando con ello el sistema espacio consecutivo para 31 componentes individuales de tipo *double*.

Las dos instrucciones anteriores se pueden unir en una sola:

```
double[] lluvia = new double[31];
```

El valor mediante el cual se define el número de elementos del array tiene que ser una expresión entera, pero no tiene por qué ser un literal como en el ejemplo anterior. El

tamaño de un array se puede establecer durante la ejecución, como en el siguiente ejemplo:

```
// usamos un array para almacenar las edades de un grupo de personas

//la vble numPersonas contiene el número de personas del grupo
int numPersonas = tec.nextInt();

...
int[] edad = new int[numPersonas];
```

2.2.- Acceso a los componentes

Como ya hemos dicho, el acceso a los componentes del array se realiza mediante subíndices. La sintaxis para referirse a un componente del array es la siguiente:

nombreVariable [subíndice]

Tras declarar el array lluvia, se dispone de 31 componentes de tipo *double* numeradas desde la 0 a la 30 y accesibles mediante la notación: lluvia[0] (componente primera), lluvia[1] (componente segunda) y así sucesivamente hasta la última componente: lluvia[30].

Con cada una de las componentes del array de *double* lluvia es posible efectuar todas las operaciones que podrían realizarse con variables individuales de tipo *double*, por ejemplo, dadas las declaraciones anteriores, las siguientes instrucciones serían válidas:

```
lluvia[0] = tec.nextDouble();
System.out.println(lluvia[0]);
lluvia[1] = lluvia[0] + 1;
lluvia[2] = lluvia[0] + lluvia[1];
lluvia[2] ++;
```

Además, hay que tener en cuenta que el **subíndice** ha de ser una **expresión entera**, por lo que también son válidas expresiones como las siguientes:

```
int i;

....
lluvia[i] = lluvia[i+1];
lluvia[i+2] = lluvia[i];
```

2.3.- Inicialización

Cuando creamos un array, Java inicializa automáticamente sus componentes:

- Con 0 cuando los componentes son de tipo numérico.
- Con false cuando los componentes son *boolean*.
- Con el carácter de ASCII 0, cuando los componentes son *char*.
- Con *null* cuando son objetos (Strings, etc)

Aun así, es probable que estos no sean los valores con los que queremos inicializar el array. Tenemos entonces dos posibilidades:

- Acceder individualmente a los componentes del array para darles valor:

```
int[] edad = new int[10];
edad[0] = 25;
edad[1] = 10;
...
edad[9] = 12;
```

- O inicializar el array en la declaración de la siguiente forma:

```
int[] edad = {25,10,23,34,65,23,1,67,54,12};
```

Es decir, enumerando los valores con los que se quiere inicializar cada componente, encerrados entre llaves. De hacerlo así, no hay que crear el array con el operador new. Java crea el array con tantos componentes como valores hemos puesto entre llaves.

2.4.- Un ejemplo práctico

Ya hemos resuelto en temas anteriores el problema de devolver el nombre de un mes dado su número. Vamos a resolverlo ahora ayudandonos de arrays:

```
public static String nombreMes(int mes){
    String[] nombre= {" ", "enero", "febrero", "marzo", "abril",
                     "mayo", "junio", "julio", "agosto",
                     "septiembre", "octubre", "noviembre", "diciembre"};

    return nombre[mes];
}
```

El método define un array de *String* que se inicializa con los nombres de los doce meses. La primera componente del array (nombre[0]) se deja vacía, de forma que enero quede almacenado en nombre[1].

Devolver el nombre del mes indicado se reduce a devolver el componente del array cuyo número indica el parámetro mes: *nombre[mes]*

2.5.- Arrays como parámetros. Paso de parámetros por referencia.

Hasta el momento sólo se ha considerado el **paso de parámetros por valor** ; de manera que cualquier cambio que el método realice sobre los parámetros formales no modifica el valor que tiene el parámetro real con el que se llama al método. En java, todos los parámetros de tipo simple (byte, short, int, ...) se pasan por valor.

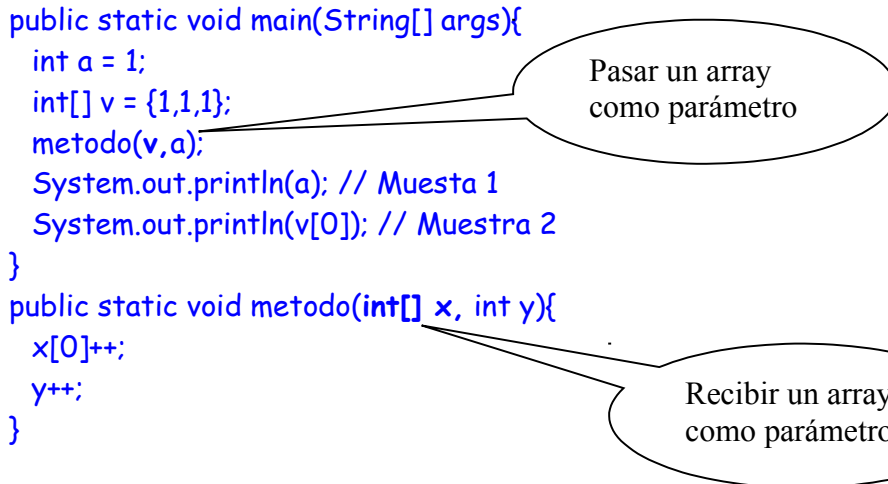
Por el contrario, los arrays no son variables de tipo primitivo, y como cualquier otro objeto, se pasa siempre **por referencia**.

En el **paso de parámetros por referencia** lo que se pasa en realidad al método es la dirección de la variable u objeto. Es por esto que el papel del parámetro formal es el de ser una referencia al parámetro real; la llamada al método no provoca la creación de una nueva variable. De esta forma, las modificaciones que el método pueda realizar sobre estos parámetros se realizan efectivamente sobre los parámetros reales. En este caso, ambos parámetros (formal y real) se pueden considerar como la misma varia-

ble con dos nombres, uno en el método llamante y otro en el llamado o invocado, pero hacen referencia a la misma posición de memoria.

En el siguiente ejemplo, la variable *a*, de tipo primitivo, no cambia de valor tras la llamada al método. Sin embargo la variable *v*, array de enteros, si se ve afectada por los cambios que se han realizado sobre ella en el método:

```
public static void main(String[] args){
    int a = 1;
    int[] v = {1,1,1};
    metodo(v,a);
    System.out.println(a); // Muestra 1
    System.out.println(v[0]); // Muestra 2
}
public static void metodo(int[] x, int y){
    x[0]++;
    y++;
}
```



Como podemos observar, para pasar un array a un método, simplemente usamos el nombre de la variable en la llamada.

En la cabecera del método, sin embargo, tenemos que utilizar los corchetes [] para indicar que el parámetro recibido es un array.

2.6.- El atributo length

Todas las variables de tipo *array* tienen un atributo **length** que permite consultar el número de componentes del array. Su uso se realiza posponiendo *.length* al nombre de la variable:

```
double[] estatura = new double[25];
....
System.out.println(estatura.length); // Mostrará por pantalla: 25
```

3.- Problemas de recorrido y búsqueda.

Muchos de los problemas que se plantean cuando se utilizan *arrays* pueden clasificarse en dos grandes grupos de problemas genéricos: los que conllevan el recorrido de un array, y los que suponen la búsqueda de un elemento que cumpla cierta característica dentro del *array*.

La importancia de este tipo de problemas proviene de que surgen, no sólo en el ámbito de los arrays, sino también en muchas otras organizaciones de datos de uso frecuente (como las listas, los ficheros, etc.). Las estrategias básicas de resolución que se verán a continuación son también extrapolables a esos otros ámbitos.

3.1.- Problemas de recorrido

Se clasifican como problemas de recorrido aquellos que para su resolución exigen algún tratamiento de todos elementos del array. El orden para el tratamiento de estos elementos puede organizarse de muchas maneras: ascendente, descendente, ascendente y descendente de forma simultánea, etc.

En el siguiente ejemplo se muestra un método en java para devolver, a partir de un array que contiene la pluviosidad de cada uno de los días de un mes, la pluviosidad media de dicho mes. Para ello se **recorren ascendente** los componentes del array para ir sumándolos:

```
public static double pluviosidadMedia(double[] lluvia){
    double suma = 0;
    //Recorremos el array
    for (int i = 0; i<lluvia.length; i++){
        suma += lluvia[i];
    }
    double media = suma / lluvia.length;
    return media;
}
```

La forma de recorrer el array ascendente es, como vemos, utilizar una variable entera (i en nuestro caso) que actúa como subíndice del array. Éste subíndice va tomando los valores 0, 1, ..., lluvia.length-1 en el seno de un bucle, de manera que se accede a todos los componentes del array para sumarlos.

El mismo problema resuelto con un **recorrido descendente** sería como sigue:

```
public static double pluviosidadMedia(double[] lluvia){
    double suma = 0;
    //Recorremos el array
    for (int i = lluvia.length-1; i>=0; i--){
        suma += lluvia[i];
    }
    double media = suma / lluvia.length;
    return media;
}
```

También realizamos un recorrido para obtener la **pluviosidad máxima del mes** (la cantidad de lluvia más grande caída en un día), es decir, el elemento más grande del array:

```
public static double pluviosidadMaxima(double[] lluvia){
    // Suponemos el la pluviosidad máxima se produjo el primer día
    double max = lluvia[0];

    //Recorremos el array desde la posición 1, comprobando si hay una
    // pluviosidad mayor
    for (int i = 1; i<lluvia.length; i++)
        if(lluvia[i] > max) max = lluvia[i];
}
```

```
    return max;
}
```

3.2.- Problemas de búsqueda

Se denominan problemas de búsqueda a aquellos que, de alguna manera, implican determinar si existe algún elemento del array que cumpla una propiedad dada. Con respecto a los problemas de recorrido presentan la diferencia de que no es siempre necesario tratar todos los elementos del array: el elemento buscado puede encontrarse inmediatamente, encontrarse tras haber recorrido todo el array, o incluso no encontrarse.

Consideremos, por ejemplo, el problema de encontrar cual fue el primer día del mes en que no llovió nada, es decir, el primer elemento del array con valor cero:

```
//Devolveremos el subíndice del primer componente del array cuyo valor es cero.
// Si no hay ningún día sin lluvias devolveremos -1
public static int primerDiaSinLluvia(double[] lluvia){
    int i=0 ;
    boolean encontrado = false ;
    while (i<lluvia.length && !encontrado){
        if (lluvia[i] == 0) encontrado = true ;
        else i++ ;
    }
    if (encontrado) return i ;
    else return -1 ;
}
```

Hemos utilizado el esquema de búsqueda: Definimos una variable *boolean* que indica si hemos encontrado o no lo que buscamos. El bucle se repite mientras no lleguemos al final del array y no hayamos encontrado un día sin lluvias.

También es posible una solución sin utilizar la variable *boolean*:

```
public static int primerDiaSinLluvia(double[] lluvia){
    int i=0 ;
    while (i<lluvia.length && lluvia[i] != 0)
        i++;

    if (i == lluvia.length) return -1 ;
    else return i;
}
```

En este caso el subíndice *i* se incrementa mientras estemos dentro de los límites del array y no encontremos un día con lluvia 0. Al finalizar el bucle hay que comprobar por cual de las dos razones finalizó: ¿Se encontró un día sin lluvias o se recorrió todo el array sin encontrar ninguno? En esta comprobación es importante no acceder al array si existe la posibilidad de que el subíndice esté fuera de los límites del array. La siguiente comprobación sería incorrecta:

```
if (lluvia[i] == 0) return i;
```

```
else return -1;
```

ya que, si se ha finalizado el bucle sin encontrar ningún día sin lluvia, *i* valdrá *lluvia.length*, que no es una posición válida del array, y al acceder a *lluvia[i]* se producirá la excepción *ArrayIndexOutOfBoundsException* (*índice del array fuera de los límites*)

Por otra parte, el mismo problema se puede resolver utilizando la sentencia *for*, como hemos hecho otras veces. Sin embargo la solución parece menos intuitiva porque el cuerpo del *for* quedaría vacío:

```
public static int primerDiaSinLluvia(double[] lluvia){
    int i;
    for (i=0; i<lluvia.length && lluvia[i] != 0; i++) /*Nada*/ ;

    if (i == lluvia.length) return -1 ;
    else return i;
}
```


Búsqueda descendente

En los ejemplos de búsqueda anteriores hemos iniciado la búsqueda en el elemento cero y hemos ido ascendiendo hasta la última posición del array. A esto se le llama **búsqueda ascendente**.

Si queremos encontrar el último día del mes en que no llovió podemos realizar una **búsqueda descendente**, es decir, partiendo del último componente del array y decrementando progresivamente el subíndice hasta llegar a la posición cero o hasta encontrar lo buscado:

```
public static int ultimoDiaSinLluvia(double[] lluvia){
    int i=lluvia.length-1;
    boolean encontrado = false ;
    while (i>=0 && !encontrado){
        if (lluvia[i] == 0) encontrado = true ;
        else i-- ;
    }
    if (encontrado) return i ;
    else return -1 ;
}
```

Búsqueda en un array ordenado: búsqueda binaria

Cuando buscamos en un array que tiene sus elementos ordenados, la búsqueda se puede optimizar.

Para ello iniciaremos la búsqueda en la posición central del array.

- Si el elemento central es el buscado habremos finalizado la búsqueda.
- Si el elemento central es mayor que el buscado, tendremos que continuar la búsqueda en la mitad izquierda del array ya que, al estar éste ordenado todos los elementos de la mitad derecha serán también mayores que el buscado.
- Si el elemento central es menor que el buscado, tendremos que continuar la búsqueda en la mitad derecha del array ya que, al estar éste ordenado todos los elementos de la mitad izquierda serán también menores que el buscado.

En un solo paso hemos descartado la mitad de los elementos del array. Para buscar en la mitad izquierda o en la mitad derecha utilizaremos el mismo criterio, es decir, iniciaremos la búsqueda en el elemento central de dicha mitad, y así sucesivamente hasta encontrar lo buscado o hasta que descubramos que no está.

Supongamos por ejemplo que, dado un array que contiene edades de personas, ordenadas de menor a mayor queremos averiguar si hay alguna persona de 36 años o no. El siguiente método soluciona este problema realizando una búsqueda binaria:

```
public static boolean hayAlguienDe36(int[] edad){
    // Las variables izq y der marcarán el fragmento del array en el que
    // realizamos la búsqueda. Inicialmente buscamos en todo el array.
    int izq = 0;
    int der = edad.length-1;

    boolean encontrado = false;
    while( izq <= der && !encontrado) {
        // Calculamos posición central del fragmento en el que buscamos
        int m = (izq + der) / 2;

        if ( edad[m] == 36)
            // Hemos encontrado una persona de 36
            encontrado = true;
        else if ( edad[m] > 36) {
            // El elemento central tiene más de 36.
            //Continuamos la búsqueda en la mitad izquierda. Es decir,
            // entre las posiciones izq y m-1
            der = m - 1;
        }
        else
            // El elemento central tiene menos de 36.
            //Continuamos la búsqueda en la mitad derecha. Es decir,
            // entre las posiciones m+1 y der
            izq = m + 1;
    } // del while
    return encontrado; // if (encontrado) return true; else return false;
}
```

La búsqueda finaliza cuando encontramos una persona con 36 años (encontrado ==true) o cuando ya no es posible encontrarla, circunstancia que se produce cuando izq y der se cruzan (izq>der)

3.- La clase Arrays.

Muchas de las operaciones que estamos habituados a realizar con otros tipos de objetos no producen el resultado esperado cuando las realizamos sobre arrays.

Por ejemplo, si queremos mostrar por pantalla el contenido de un array double[] lluvia, no sería adecuado ejecutar:

```
System.out.pritln(lluvia.toString());
```

Si queremos ver si dos arrays `int[] a` e `int[] b` contienen los mismos elementos, tampoco serviría ejecutar:

```
if (a.equals(b)) ...
```

Ello se debe a que la clase a la que pertenecen los arrays no tiene reescritos los métodos `equals` y `toString` para que hagan la operación esperada.

Para dar solución a estas necesidades disponemos de la **clase Arrays**. Esta clase contiene métodos capaces de realizar operaciones con arrays completos, como por ejemplo devolver el contenido del array como un `String`, comparar el contenido de dos arrays, ordenar, realizar búsquedas dentro del array, ... A continuación enumeramos algunos métodos útiles de la clase `Arrays`:

- `Arrays.equals(xxx[] a, xxx[] b)`: Devuelve `true` si los arrays `a` y `b` contienen los mismos elementos en el mismo orden. `xxx` sería el tipo de los elementos del array (`int`, `byte`, `double`, etc)
- `Arrays.fill(xxx[] a, xxx b)`: Rellena todo el array `a` con el valor `b`.
- `Arrays.sort(xxx[] a)`: Ordena el array `a` en orden ascendente. Si los elementos del array son objetos, el criterio de ordenación será el que determine el método `compareTo()` de la clase a la que pertenece el objeto. La clase a la que pertenecen los objetos del array tienen que implementar el interface `Comparable`. Los interfaces se estudiarán en un tema posterior.
- `Arrays.toString(xxx[] a)`: Devuelve un `String` que contiene a cada uno de los elementos del array separados por comas y en orden ascendente.

4.- Listas : *ArrayList*

Al igual que los arrays, las Listas (`List`) son una colección de datos del mismo tipo. Se dice que es una estructura de datos lineal porque en ellas, hay un primer elemento, un último elemento y cada elemento tiene un elemento antes y otro después de él (excepto el primero y el último, claro).

¿Qué diferencias fundamentales hay entonces entre una Lista y un array ?:

- Los elementos de una Lista, en Java, tienen que ser objetos. No pueden ser datos de tipo primitivo. Así, por ejemplo, no se puede crear una Lista de datos de tipo `int`, aunque si se podría crear una Lista de `Integer` (`Integer` es una de las clases llamadas " envoltorio ". Sería como una versión orientada a objetos del dato primitivo `int`).
- **Cuando se crea una Lista no se especifica su tamaño**. Inicialmente la lista no tiene elementos (tamaño cero) y su longitud irá creciendo a medida que se añaden elementos a ella.
- Al añadir elementos a la Lista podemos indicar qué posición exacta queremos que ocupe el nuevo elemento dentro de la lista. Es decir, que podemos **insertar** elementos en cualquier posición de la lista. El elemento que ocupaba anteriormente la posición indicada (y todos sus siguientes) se verán desplazados una posición a la derecha.
- Es posible eliminar de la lista el elemento que ocupa determinada posición. En tal caso, los elementos que había a continuación se desplazarán una posición a la izquierda.

El API Collections de Java tiene varias clases que representan una Lista. Una de ellas es la clase **ArrayList**. Un ArrayList es una clase que tiene los métodos típicos de este tipo de colección (Lista) pero que, internamente está implementada utilizando arrays. En temas posteriores veremos otras clases que implementan Listas, en concreto, la clase **LinkedList**.

4.1.- La clase ArrayList

Declaración y creación de un ArrayList :

```
ArrayList<Tipo> nombreVariable = new ArrayList<>() ;
```

Ejemplos :

```
ArrayList<String> nombres = new ArrayList<>() ;
ArrayList<Figuras> dibujo = new ArrayList<>() ;
ArrayList<Object> elementos = new ArrayList<>() ;
```

Como vemos, en la declaración de la variable indicamos el tipo de los elementos que van a formar parte de la Lista, encerrandolo entre <> (« Notación de diamante »). Esta notación está disponible a partir de la versión 1.7 de Java. Anteriormente no se especificaba el tipo de los elementos de la Lista y ésta podía contener cualquier tipo de Object :

```
ArrayList nombres = new ArrayList() ;
```

Algunos métodos interesantes

De consulta		
boolean	clear()	Elimina todos los elementos de la Lista. La vacía. Devuelve true si la Lista cambia su contenido.
boolean	contains(element)	Devuelve true si la Lista contiene el elemento especificado
Elemento	get(indice)	Devuelve el elemento de la lista que ocupa la posición indicada.
int	indexOf(elemento)	Devuelve la posición que ocupa en la lista el elemento indicado (primera aparición). Devuelve -1 si no se encuentra.
int	lastIndexOf(elemento)	Devuelve la posición que ocupa en la lista el elemento indicado (última aparición). Devuelve -1 si no se encuentra.
int	size()	Devuelve el número de elementos de la lista.
De modificación		
boolean	add(elemento)	Añade un elemento al final de la lista. Devuelve true si la lista cambia su contenido
void	add(posición, elemento)	Añade un elemento en la posición especificada de la lista. El que ocupaba dicha posición y los pos-

		teriores quedarán desplazados a la derecha. Devuelve true si la lista cambia su contenido
boolean	addAll(colección)	Añade todos los elementos de la colección (por ejemplo otra Lista), al final de la Lista. Devuelve true si la lista cambia su contenido
boolean	addAll(posición,colección)	Añade todos los elementos de la colección en la posición especificada de la lista. El que ocupaba dicha posición y los posteriores quedarán desplazados a la derecha. Devuelve true si la lista cambia su contenido
Elemento	remove(indice)	Elimina el elemento de la posición indicada. Devuelve el elemento eliminado
boolean	remove(elemento)	Elimina el elemento indicado, si existe. Devuelve true si la lista cambia su contenido. ^o

Recorrido.

Vamos a ver tres formas de recorrer un ArrayList

Sea el ArrayList :

```
ArrayList<String> lista = new ArrayList<>();
```

1. Con un bucle, utilizando el método size() y el método get(indice)

```
for(int i = 0 ; i< lista.size() ; i++){
    System.out.println(lista.get(i));
}
```

2. Con un bucle « foreach »

```
for(String s : lista){
    System.out.println(s);
}
```

3. Con un objeto Iterator (Forma preferida)

```
Iterator<String> it = lista.iterator();
while(it.hasNext()){
    System.out.println(it.next());
}
```

4.- Arrays bidimensionales: matrices

Los arrays bidimensionales, también llamados matrices, son muy similares a los arrays que hemos visto hasta ahora: También son una colección de elementos del mismo tipo que se agrupan bajo un mismo nombre de variable. Sin embargo:

- Sus elementos están **organizados en filas y columnas**. Tienen, por tanto una altura y una anchura, y por ello se les llama bidimensionales.
- A cada componente de una matriz se accede mediante dos subíndices: el primero se refiere al número de fila y el segundo al número de columna. En la siguiente figura, $m[0][0]$ es 2, $m[0][3]$ es 9, $m[2][0]$ es 57

m		columna			
fila		0	1	2	3
	0	2	5	6	9
	1	3	2	2	8
	2	57	12	15	36
	3	33	6	12	6
	4	0	41	5	7

- Como vemos, filas y columnas se numeran a partir del 0.

Si se quisiera extender el tratamiento el estudio de la pluviosidad, para abarcar no solo los días de un mes sino los de todo un año, se podría definir, por ejemplo, un array de 366 elementos, que mantuviera de forma correlativa los datos de pluviosidad de una zona día a día. Con ello, por ejemplo, el dato correspondiente al día 3 de febrero ocuparía la posición 34 del array, mientras que el correspondiente al 2 de julio ocuparía el 184.

Una aproximación más conveniente para la representación de estos datos consistiría en utilizar una matriz con 12 filas (una por mes) y 31 columnas (una por cada día del mes). Esto permitiría una descripción más ajustada a la realidad y, sobre todo, simplificaría los cálculos de la posición real de cada día en la estructura de datos. El elemento $[0][3]$ correspondería, por ejemplo, a las lluvias del 4 de enero.

4.1.- Matrices en Java

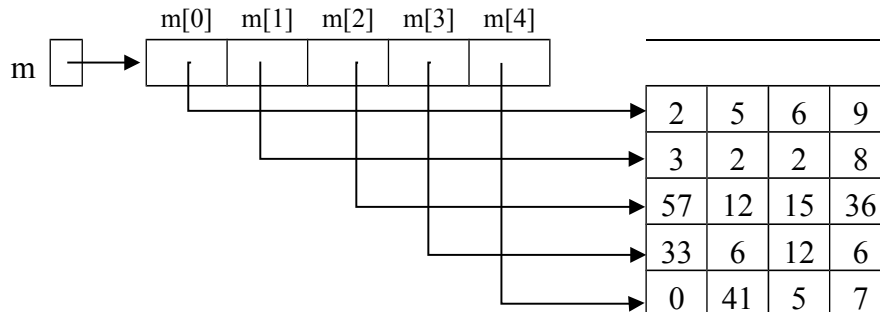
En Java, una matriz es, en realidad un array en el que cada componente es, a su vez, un array. Dicho de otra manera, una matriz de enteros es un array de arrays de enteros.

Esto, que no es igual en otros lenguajes de programación, tiene ciertas consecuencias en la declaración, creación y uso de las matrices en Java:

- Una matriz, en Java, puede tener distinto número de elementos en cada fila.
- La creación de la matriz se puede hacer en un solo paso o fila por fila.
- Si m es una matriz de enteros ...
 - $m[i][j]$ es el entero de la fila i , columna j
 - $m[i]$ es un array de enteros.
 - $m.length$ es el número de filas de m .

- `m[i].length` es el número de columnas de la fila `i`

Podríamos dibujar la matriz `m` del ejemplo anterior de una forma más cercana a cómo Java las representa internamente:



4.2. Declaración de matrices.

El código siguiente declara una matriz (array bidimensional) de elementos de tipo `double`, y la crea para que tenga 5 filas y 4 columnas (matriz de 5x4):

```
double[][] m = new double[5][4];
```

La siguiente declaración es equivalente a la anterior aunque en la práctica es menos utilizada a no ser que queramos que cada fila tenga un número distinto de elementos:

```
double[][] m = new double [5][];
m[0] = new double[4];
m[1] = new double[4];
m[2] = new double[4];
m[3] = new double[4];
m[4] = new double[4];
```

Es posible inicializar cada uno de los subarrays con un tamaño diferente (aunque el tipo base elemental debe ser siempre el mismo para todos los componentes). Por ejemplo:

```
double[][] m = new double [5][];
m[0] = new double[3];
m[1] = new double[4];
m[2] = new double[14];
m[3] = new double[10];
m[4] = new double[9];
```

4.3. Inicialización.

```
int[][] m = {
    {7,2,4},
```

```
        {8,2,5},  
        {9,4,3},  
        {1,2,4}  
    };
```

7	2	4
8	2	5
9	4	3
1	2	4

4.3. Recorrido

El recorrido se hace de forma similar al de un array aunque, dado que hay dos subíndices, será necesario utilizar dos bucles anidados: uno que se ocupe de recorrer las filas y otro que se ocupe de recorrer las columnas.

El siguiente fragmento de código recorre una matriz m para imprimir sus elementos uno a uno.

```
    for(int i = 0 ; i < m.length ; i++)  
        for(int j = 0; j < m[i].length; j++)  
            System.out.println(m[i][j])
```

El recorrido se ha hecho por filas, es decir, se imprimen todos los elementos de una fila y luego se pasa a la siguiente. Como habíamos indicado anteriormente, m.length representa el número de filas de m, mientras que m[i].length el número de columnas de la fila i

También es posible hacer el recorrido por columnas: imprimir la columna 0, luego la 1, etc:

```
int numFilas = m.length;
int numColumnas = m[0].length;
for(int j = 0 ; j < numColumnas; j++)
    for(int i = 0; i < numFilas; i++)
        System.out.println(m[i][j])
```

o, directamente ...

```
for(int j = 0 ; j < m[0].length ; j++)
    for(int i = 0; i < m.length; i++)
        System.out.println(m[i][j])
```

En este caso, para un funcionamiento correcto del recorrido sería necesario que todas las columnas tuvieran igual número de elementos, pues en el bucle externo, se toma como referencia para el número de columnas la longitud de m[0], es decir el número de elementos de la primera fila.

5.- Arrays multidimensionales

En el punto anterior hemos visto que podemos definir arrays cuyos elementos son a la vez arrays, obteniendo una estructura de datos a la que se accede mediante dos subíndices, que hemos llamado arrays bidimensionales o matrices.

Este “anidamiento” de estructuras se puede generalizar, de forma que podríamos construir arrays de más de dos dimensiones. En realidad Java no pone límite al número de subíndices de un array. Podríamos hacer declaraciones como las siguientes:

```
int[][][] notas = new int[10][5][3]; //Notas de 10 alum. en 5 asign. en 3 eval.
double[][][][] w = new double [2][7][10][4][10];
```

Sin embargo, encontrar ejemplos en los que sean necesarios arrays de más de tres dimensiones es bastante raro, y aún cuando los encontramos solemos utilizar arrays de uno o dos subíndices porque nos resulta menos complejo manejarlos.