Capítulo 8

ESTRUCTURAS ESTÁTICAS DE DATOS: ARRAYS

1. INTRODUCCIÓN

En el capítulo 4 se estudiaron los tipos de datos básicos de Java y el mecanismo que permite definir y usar variables simples donde almacenar datos de esos tipos. En muchas aplicaciones, el programador necesita procesar una cantidad suficientemente elevada de datos del mismo tipo que están relacionados de alguna forma. Con los conocimientos adquiridos hasta el momento, sólo se dispone de variables simples, por lo que debería utilizar un número también elevado de ellas para almacenar dichos datos. Por ejemplo, imagine que se desea procesar los precios de los platos de un restaurante para obtener una lista ordenada de ellos. Habría que guardar cada uno de ellos en una variable y la tarea de ordenarlos sería, como puede suponer, extremadamente complicada. Se hace necesaria una nueva forma de almacenar datos que pueda solucionar este problema.

En todos los lenguajes de programación se pueden definir estructuras de datos complejas, que guardan más de un dato y que son fundamentales para un programador. Conocerlas, saber usarlas y elegir las más adecuadas para cada algoritmo, hará que los programas sean mucho más sencillos y eficientes. Nicklaus Wirth, informático eminente y padre del lenguaje Pascal, tituló su obra fundamental sobre programación con la ecuación *Algoritmos + Estructuras de datos = Programas*, destacando así la importancia de las estructuras de datos como pilar fundamental de la programación.

En este capítulo, se estudiará la forma en que datos del mismo tipo pueden agruparse, formando conjuntos de elementos organizados bajo un nombre común que los referencia, los *arrays*. Son conocidos también como *tablas*, *vectores* o, incluso, *arreglos* en algunas traducciones del inglés array.

2. CONCEPTO DE ESTRUCTURA DE DATOS

En la resolución informática de algoritmos, a veces surge la necesidad de expresar las relaciones que existen entre distintas variables, o bien almacenarlas y referirlas como si de un grupo se tratara. Por ejemplo, para referenciar a todos los alumnos de primero del ciclo de Desarrollo de Aplicaciones Multiplataforma, se puede usar el nombre *Primero de DAM*, en lugar de nombrar a cada alumno en particular. Como bien puede comprenderse, es muy complicado realizar algoritmos sobre un conjunto de elementos si cada uno de ellos se referencia por una variable. Siguiendo con nuestro ejemplo, si se desea efectuar un listado ordenado de notas de la asignatura de Programación de los alumnos del grupo, deberán guardarse todas para su proceso. Si se utiliza una variable para guardar la nota de cada alumno, será complicado procesarlas para realizar el listado. Para solucionar este problema, aparecen las llamadas *estructuras de datos*, entendidas como un aglomerado de elementos caracterizado por un método de estructuración particular.

Se puede dar una definición formal de **estructura de datos** como un conjunto de componentes cuya organización se caracteriza por las funciones de acceso que se usan para almacenar y suprimir elementos individuales del grupo.

Por ejemplo, suponga que se dispone de un ente de tipo "*ladrillo*" que permite formar una estructura con ellos llamada "*pared*". Como definición de la estructura pared puede decirse que es un "conjunto de ladrillos colocados en filas en posición vertical, de forma que en la primera fila están situados horizontalmente uno a continuación de otro y las restantes se forman disponiendo uno encima de dos de la fila anterior. La última fila no tiene ladrillos encima". Esta estructura tiene sus propias funciones de acceso:

- Almacenar: poner un elemento significa que deben estar puestos los dos de abajo, sobre los cuales se apoya, salvo los elementos que ocupan la fila que pega al suelo, en cuyo caso se coloca sin más.
- Suprimir: para quitar un elemento de esta estructura es imprescindible haber quitado con anterioridad los dos que tiene encima, salvo que el elemento pertenezca a la última fila, en cuyo caso se quita sin más.

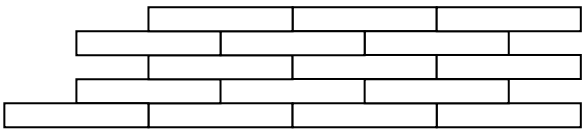


Figura 7.1. Estructura de datos "pared"

Otro ejemplo podría ser el tipo "bola de bingo" y la estructura "bombo con bolas de bingo", sobre la que se definen las funciones de acceso. Para almacenar un elemento se irá metiendo en el bombo, una a una, todas las bolas del conjunto, cada vez que se desee comenzar un juego, y para suprimir o recuperar un elemento bastará con sacar una bola.

Como puede observar, la definición de estructura no especifica como se realiza el modo de acceso, sólo que las funciones acceden al elemento deseado. El programador no tiene por qué conocer cómo se implementa el acceso en las estructuras de datos suministradas por los lenguajes de programación, por lo que se dice que el acceso es transparente para él. En cuanto a las estructuras de datos que no suministran los lenguajes de programación, es responsabilidad del programador, que no sólo debe definir la estructura, además, debe especificar e implementar el conjunto de subprogramas encargados de realizar el acceso para cada estructura en particular.

Las estructuras de datos suministradas en Java son: Arrays, Enumeraciones, Ficheros y Clases. Aunque si bien no proporciona otro tipo de estructuras de datos si implementa clases de muy diversas características y propósitos, con distintas implementaciones de las estructuras de datos nombradas anteriormente. Implementaciones de arrays como: Array, ArrayDeque, ArrayList, Arrays, Vector (crece y decrece a medida), y también implementaciones tales como List, Stack, Queue, JTree, TreeNode, por nombrar algunas de ellas.

2.1. Abstracción de datos

Se estudió en el capítulo 2 que los datos se representan físicamente en memoria de distintas formas, dependiendo en primer lugar de su tipo. Pero, aún tratándose del mismo tipo, pueden representarse también con distintos métodos. Por ejemplo, los enteros pueden estar representados en una máquina en binario puro, en otra en módulo y signo, en una tercera en complemento a 1, etc. Aunque se conoce en que consisten tales métodos de representación a nivel teórico, generalmente, no se sabe nada acerca de como están representados los enteros en la máquina que se usa habitualmente, lo cual no impide utilizarlos en programas como java o cualquier otro.

Por otro lado, la implementación de las operaciones sobre los enteros depende directamente de la representación interna de los mismos. Como programadores tampoco es necesario conocer este nivel de detalle, simplemente se utilizan operaciones sobre enteros en los programas. Se usan sentencias de declaración de variables de tipo entero y se conocen las operaciones que se pueden realizar sobre ellas en Java, suma, resta, multiplicación, división, hallar el resto, asignación, incremento y decremento y las operaciones relacionales y lógicas. Estas operaciones están incorporadas por el lenguaje y están diseñadas de forma tal que permite al programador crear, almacenar, recuperar o modificar un dato, abstrayéndolo de la tarea de diseñarlas. Se puede decir que Java ha *rodeado* o *encapsulado* al tipo entero con un paquete de operaciones para manejarlos, cuya implementación es absolutamente transparente para el programador, que conociendo su descripción lógica sólo debe preocuparse de utilizarlos correctamente.

En este contexto podemos definir la **abstracción de datos** como la separación que existe entre la representación interna de los mismos y sus funciones de acceso por un lado y, por otro, las aplicaciones que utilizan esos datos a nivel lógico.

Por otro lado el *encapsulamiento de datos* en este contexto consiste en encerrar la definición de la estructura de datos y la implementación de sus funciones de acceso en una “cápsula” donde permanecen ocultas para el usuario.

Las estructuras de datos no suministradas por el lenguaje que se desea utilizar en programas de aplicación, no sólo hay que definirlas a nivel lógico, sino que, además, es obligado implementar las funciones para su manipulación.

2.2. Clasificación de las estructuras de datos

Las estructuras de datos se dividen en dos tipos en función de cómo utilizan la memoria:

ESTRUCTURAS DE DATOS ESTÁTICAS

Son aquellas que ocupan una cantidad fija de memoria en tiempo de compilación.

En el ejemplo que se citó anteriormente, la estructura "pared" formada con tipo "ladrillo", suponga que se almacenan 500 ladrillos para levantar una pared y que, en principio, se desconoce cuantos ladrillos se utilizarán en su construcción. En este caso, los 500 ladrillos ocupan un espacio fijo de almacenamiento físico.

ESTRUCTURAS DE DATOS DINÁMICAS

Son aquellas que utilizan una cantidad variable de memoria en función de los requerimientos en tiempo de ejecución. A este grupo pertenecen todas las estructuras no facilitadas habitualmente por los lenguajes de propósito general mencionadas anteriormente, listas, pilas, colas,.... Sin embargo, el lenguaje aporta todas las herramientas necesarias para que sean incorporadas por el programador, que en este caso debe implementar el tipo y las funciones para su manipulación. A veces son suministradas las clases que implementan estas estructuras, como es el caso de Java.

Siguiendo con el mismo ejemplo, suponga ahora que existe la posibilidad de construir la pared a demanda de ladrillo, es decir, se pide ladrillo y se coloca, se pide ladrillo y se coloca, y así, sucesivamente, hasta acabar la pared. No ha habido almacenamiento estático de ladrillos, sino que se ha ido usando dinámicamente lo que se ha necesitado.

3. PERSPECTIVAS DE ANÁLISIS DE UNA ESTRUCTURA DE DATOS

En la definición de estructura de datos, no se dice nada del significado de las relaciones entre los elementos de la estructura. Se habla de la relación física o lógica de los elementos, o la forma de acceso, pero no del significado de estas relaciones, que sólo puede conocerse cuando se use la estructura en un programa, para representar los datos de un problema en particular.

Por ejemplo, ¿qué significado tiene la pared del ejemplo anterior? Si con esa estructura se ha construido un plano de un habitáculo, no tendría significado alguno, hasta que sea destinado a algo en particular: dormitorio, aula, oficina, gimnasio, etc.

Existen tres formas distintas de observar una estructura de datos:

- Como una colección abstracta de elementos con un conjunto de funciones de acceso, es decir, desde la definición teórica de la estructura.

- Como un problema de codificación, ¿cómo se implementan las funciones de acceso?

- Como una forma de representar las relaciones entre datos en un contexto especifico, es decir, la manera de usarla en un caso concreto.

Estos tres puntos de mira dan lugar a tres niveles en el análisis de la estructura, abstracto, de implementación y de uso.

NIVEL ABSTRACTO O LÓGICO

Es aquél en el que se define la organización de los datos y se especifican los módulos generales de acceso. En este nivel se desconoce si estos módulos de acceso serán diseñados como funciones o como procedimientos, tan sólo se expone el proceso que realizan. Por ejemplo, si se desea una estructura de tipo *cola* se puede definir como usa sucesión de elementos, de manera que almacenar un elemento consiste en ponerlo a continuación del último que llegó y sacar un elemento supone sacar a los que tiene delante, es decir, el primero que llega es el primero que sale.

NIVEL DE IMPLEMENTACIÓN

En este nivel se estudian las formas de representación de los datos en memoria, y se implementan las funciones y procedimientos de acceso en un lenguaje de codificación en particular. Se examinará detenidamente las distintas formas en que pueden implementarse en dicho lenguaje las estructuras de datos definidas en el nivel anterior y se escogerá la más adecuada. Por ejemplo, una estructura de datos de tipo *cola* puede ser implementada en Java mediante un array o dinámicamente mediante una lista enlazada. Cuál es la implementación más conveniente será una cuestión a decidir por el programador.

NIVEL DE APLICACIÓN O USO

A este nivel corresponde el uso de la estructura, definida e implementada, en un contexto determinado. Puede decirse que a los niveles de aplicación corresponden los ejemplos específicos relativos a los otros dos niveles. Por ejemplo, si se ha implementado la estructura de tipo *cola* puede utilizarse para gestionar una cola de impresión.

Veamos con un ejemplo de la vida real, una biblioteca, lo que significan estos diferentes puntos de vista. La estructura de datos “biblioteca” está compuesta por una colección particular de elementos “libros”:

- En el nivel abstracto, tratamos preguntas sobre ¿qué?: por ejemplo, ¿qué se entiende por una biblioteca?, ¿qué servicios realiza?. La estructura puede definirse de forma abstracta como una “colección de libros”, para los que se definen, por ejemplo, las siguientes operaciones o funciones de acceso:

- Incluir un libro: meter un libro en la colección.
- Sacar un libro: sacar un libro de la colección: para prestarlo al cliente, para consulta en sala, para descatalogar.
- Comprobar si se dispone de un libro: consiste en mirar el catálogo de libros de la biblioteca y comprobar si existe en dicho catálogo el libro buscado.

Cuando se define la estructura a nivel lógico no es importante saber como están organizadas las estanterías, ni cómo gestiona el bibliotecario el servicio.

- Desde el punto de vista de la implementación, se atiende a preguntas relativas al ¿cómo?: ¿cómo están catalogados los libros? ¿cómo están organizados en las estanterías? ¿cómo procede el bibliotecario para prestar un libro?. Todas estas funcionalidades deben quedar formalmente detalladas.

- En el nivel de aplicación, el usuario de biblioteca observa entidades como la Biblioteca Nacional, Biblioteca de la Universidad de Sevilla, Biblioteca de la Escuela de Ingeniería Informática, etc., que obedecen a lo descrito en el nivel lógico y tienen detalladas todas ellas el mismo proceder en los servicios que ofrecen.

El objetivo de este método de diseño es ocultar el nivel de implementación al usuario, de manera que con la definición lógica de la estructura sepa proceder cuando la usa en un caso concreto. Si se dibuja una pared para separar el nivel de aplicación o uso del nivel de implementación, tal como se muestra en la figura 8.1, cabe preguntarse ¿cómo se comunican ambos niveles?. En el ejemplo de la biblioteca la pregunta sería: ¿cómo pueden comunicarse bibliotecario y usuario de la biblioteca?. La respuesta es clara, entre ambos existe un protocolo de comunicación que no sólo consiste en el lenguaje, sino también en la forma de pedir un libro, normalmente rellenando una ficha, o la forma de devolverlo, etc. Dicho formalmente, ambos niveles se comunican gracias a la abstracción y la encapsulación.

La visión abstracta de la estructura suministra las especificaciones de las operaciones de acceso sin decir como están diseñadas esas operaciones internamente, o sea, dice qué hacen y en qué condiciones, pero no cómo. La comunicación desde el nivel de usuario hacia el de implementación se realiza teniendo en cuenta las **especificaciones de entrada, suposiciones permitidas** o **precondiciones** de las funciones de acceso que el primero debe cumplir. La comunicación desde el nivel de implementación hacia el de uso se realiza a través de las **especificaciones de salida** o **postcondiciones** de las funciones de acceso que lanzan al exterior las trasformaciones producidas sobre la estructura de datos y que el usuario debe conocer. En resumen, el nivel abstracto encapsula la estructura de datos, pero suministra vías de comunicación a través de las especificaciones de las operaciones de acceso. Esto es lo que se conoce como protocolo o **interfaz** de comunicación.

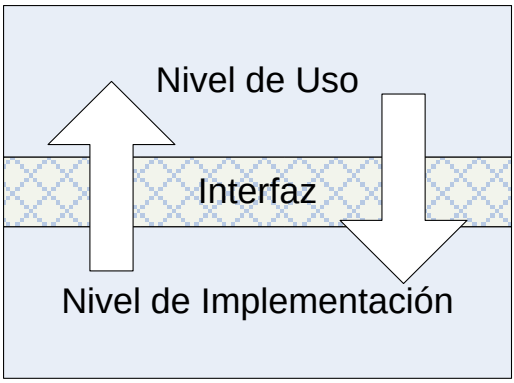


Figura 7.2. Comunicación entre el nivel de aplicación y el nivel de implementación

En resumen, el nivel de uso es independiente del nivel de implementación, de manera que la implementación de una determinada estructura de datos puede cambiar, sin que por ello afecte al uso de la estructura en programas específicos, siempre que la interfaz de comunicación se mantenga intacta en las distintas implementaciones.

4. ARRAYS UNIDIMENSIONALES

Como se ha dicho anteriormente, el array es una estructura de datos incorporada en Java y en la mayoría de los lenguajes de propósito general. Considerando las distintas formas de observar una estructura de datos, estudiadas en el apartado anterior, se analizará a continuación la estructura array desde las tres perspectivas.

4.1. Nivel Lógico

Los arrays se utilizan como **contenedores** para almacenar datos relacionados. Desde el punto de vista abstracto se define array unidimensional como una estructura de datos formada por una colección finita de elementos homogéneos, ordenados, que se referencian con un nombre común. *Homogéneo* quiere decir que todos los elementos son del mismo tipo de dato, *ordenados* indica que hay un primer elemento, un segundo elemento, etc. lo cual no indica que el valor de esos elementos esté en orden y *finito* significa que también hay un último elemento. Así definida, esta colección de elementos forma una estructura estática, lo que supone que el tamaño del array deberá ser conocido en tiempo de compilación. El hecho de ocupar un tamaño fijo en memoria no significa que todas sus casillas deban contener elementos significativos.

Por ejemplo, puede ser un array una estructura de datos para guardar las temperaturas mínimas de los últimos treinta días o los valores de las acciones de una empresa durante la última semana

La definición formal de la función de acceso de un array unidimensional es la siguiente: el mecanismo de acceso a un array es directo, es decir, puede accederse a cualquier elemento sin necesidad de acceder primero a los elementos que le preceden. El acceso a un elemento específico del array se realiza mediante el nombre del array y un índice que permite especificar cuál es el elemento deseado, dando su posición relativa dentro de la colección, el primero, el segundo o el enésimo.

¿Qué operaciones se definen para los arrays? Si Java no tuviera incorporada la estructura de datos *array*, correspondería al programador la responsabilidad de definir las funciones de acceso. Por ejemplo, habría que especificar al menos dos operaciones, *ConstruirArray* y *AccederElemento*, la primera para determinar la forma de almacenar los datos en memoria y la segunda para establecer como se recuperan elementos de la estructura. Sin embargo, como los arrays son estructuras incorporadas, es el propio lenguaje quien suministra la forma especial de realizar estas operaciones, de manera que la sintaxis y la semántica del lenguaje de codificación en particular especificarán la función de acceso. La operación de construir el array es parte de la sección de declaración de variables y objetos en general, dentro de un programa en Java. La sintaxis de Java suministra un constructor de tipo para crear arrays, así como una forma de acceder directamente a un elemento cualquiera del grupo.

Se estudiará a continuación la sintaxis y la semántica de la función de acceso a elementos de la estructura:

SINTAXIS DE LA FUNCIÓN DE ACCESO

Cada elemento puede ser accedido directamente mediante la siguiente expresión:

nombreArray [expresión del índice]

nombreArray es un identificador que representa a la colección de objetos.

expresión del índice es un valor comprendido entre 0 y *n-1*, siendo *n* el número de elementos del array. La expresión del índice señala el orden, siendo 0 el primer elemento y *n-1* el último.

SEMÁNTICA DE LA FUNCIÓN DE ACCESO

Utilizar la expresión de la función de acceso significa localizar el elemento asociado con la *expresión del índice*, en la colección de elementos cuyo nombre es *nombreArray*. La manera de referenciar un elemento se conoce como *indexación*.

La función de acceso se usa de dos formas:

1. Para especificar un lugar donde ha de copiarse un valor, es decir, se accede para almacenar.

   Ejemplos:

```
notas[2]=10   /*notas es el nombre de la colección y 2
               lugar donde se almacena 10 */
Leer (notas[5]) /*Se almacena en el 6º lugar el valor
                 leído por teclado */
notas[i]=0 //i referencia el elemento i-ésimo del array
```

2. Para especificar un lugar de donde ha de extraerse un valor, es decir, recuperar el valor del elemento.

   Ejemplos:

```
var=notas[2] /*Asigna el valor del elemento 3 del grupo
              a la variable var*/
/*Visualiza en pantalla el valor del elemento 6*/
 Escribir("La nota es: ---", notas[5])
```

REPRESENTACIÓN GRÁFICA

Puesto que los elementos de un array se almacenan en posiciones consecutivas de memoria, se puede representar gráficamente un array de *n* elementos como una tabla de *n* casillas:

Array de nombre *nombreArray*

índice	elementos
[0]	Primer elemento
[1]	Segundo elemento
[2]	Tercer elemento
....	.....
....	.....
....	.....
[n-1]	Último elemento

4.2. Nivel de Implementación: arrays en Java

La implementación de una estructura incorporada en un lenguaje de programación es transparente al usuario. Sin embargo, es necesario saber que han de realizarse, imprescindiblemente, dos tareas para la implementación de cualquier estructura de datos:

a) Reservar memoria para la estructura.

b) Codificar las funciones de acceso.

El Lenguaje de Programación JAVA maneja los arrays de manera bastante similar a como lo hacen los otros Lenguajes, pero con algunas diferencias. La más importante es que en JAVA los arrays son **objetos**.

DECLARACIÓN

La manera de reservar memoria para esta estructura de datos es mediante la sentencia de declaración, que dice al compilador cuantas casillas se requieren para representar el array, así como, el nombre de la estructura. De esta forma, el nombre del array se asocia con las características del mismo. Como está incorporada la estructura en Java tan sólo tenemos que saber cuál es la sintaxis de declaración.

Formato de declaración en Java de variables de tipo Array:

tipo nombreArray [];

o bien,

tipo [] nombreArray; //más usado

tipo: puede ser cualquier tipo de datos básico o definido por el programador, incluidos objetos. El tipo de dato Array es, a su vez, **un objeto**.

nombreArray: es un identificador que referencia a la estructura completa.

Como cualquier otro tipo de variable, la declaración no crea el array, solo dice al compilador que esta variable contendrá un array del tipo especificado.

CREACIÓN

Los arrays se crean en Java con el operador new.

nombreArray = new tipo[tamaño];

tamaño: representa la cantidad de elementos de la colección, debe evaluarse como una constante entera positiva. Los corchetes son obligatorios, forman parte de la función de acceso. Significan que el objeto declarado es un array. En Java, el tamaño del array debe conocerse en tiempo de compilación y no puede ser cambiado en ejecución.

Tipo: debe coincidir con el tipo con el que se ha declarado **nombreArray**.

nombreArray: debe ser una variable declarada como **tipo[]** anteriormente.

```
notas = new int[100];
```

Se puede declarar y crear el array en la misma instrucción de declaración.

Ejemplo

```
double[] notas = new double[tam];
```

Si el array no es creado, el compilador mostrará el siguiente mensaje de error en compilación:

```
ArrayDemo.java:4: Variable notas may not have been
initialized.
```

En resumen, el Array queda definido por

- el tipo de elementos que agrupa,
- el número de ellos, es decir, su longitud.

CARACTERÍSTICAS DE LOS ARRAYS DE UNA DIMENSIÓN

Con la declaración de un array, el nombre se asocia a las características del mismo. En Java y en la mayoría de los lenguajes que incorporan esta estructura de datos, estas características incluyen:

1. La *cota superior* del rango del índice. En Java, la cota superior es n-1.
2. La *cota inferior* del rango del índice. En Java, la cota inferior es 0.
3. La posición de memoria de la primera casilla del array llamada *dirección base*. En Java, está representada por el nombre del array sin más.
4. El número de posiciones de memoria necesarias para almacenar cada elemento del array. Siendo *tamañoTipo* los bytes que ocupa el tipo de dato del array, el compilador guardará tantos bytes como *tamañoTipo* por el número de casillas. Esto va a depender de la máquina, del tipo de datos, del lenguaje usado, etc. En cualquier caso, será el compilador quién determine la cantidad de bytes en función del tipo de datos de los elementos del array.

FUNCIONAMIENTO INTERNO

La información sobre las características del array la almacena el compilador, de manera que cuando se accede a un elemento, ya sea para almacenar o recuperar, la función de acceso utiliza esta información para localizar el elemento deseado.

En Java, el array se almacena en posiciones consecutivas de memoria, siendo la más baja para el primer elemento y la más alta para el último.

Por ejemplo, suponga que las direcciones base para los arrays siguientes son 100 y 2000 en hexadecimal.

```
float precios[40]; //Tamaño del tipo: 4 bytes por elemento
short notas[10];  // Tamaño del tipo: 2 bytes por elemento
```

	precios	notas
Cota superior	39	9
Cota inferior	0	0
Base	100 H	2000 H

Gráficamente, los arrays se pueden representar del modo siguiente:

Indice	precios	Dir Hex
		100
[0]	23.59	
		103
		104
[1]	100.20	
		107
		108
[2]	70.35	
		111
.....	.....	.....
		256
[39]	500.0	
		259

Indice	notas	Dir Hex
		2000
[0]	10	
		2001
		2002
[1]	7	
		2003
		2004
[2]	5	
		2005
.....	.....	.....
		2018
[9]	3	
		2019

Teniendo en cuenta toda esta información, se calcularán algunos datos necesarios para entender como funciona la función de acceso en Java.

El tamaño en bytes de un array se calcula de la siguiente manera:

TamañoBytes = TamañoTipo * LongitudArray.

En los ejemplos anteriores:

precios [40] ocupa 4 * 40 = 160 bytes

notas [10] ocupa 2 * 10 = 20 bytes

Un elemento cualquiera que ocupa el lugar *Indice* se localiza utilizando la siguiente fórmula:

Dirección de nombreArray [Indice] = Base + IndiceElemento * TamañoTipo

Si se desea acceder al elemento 2 del array *notas* su dirección será:

Dirección de notas [2] = 2000 + 2*2 = 2004 en hexadecimal.

Puede comprenderse que según las características del array y utilizando las fórmulas anteriores, el compilador accede a cualquier elemento del array.

COMPROBACIÓN DE RANGO

En Java, los límites superior e inferior de un array, a diferencia de otros lenguajes, se usan para controlar errores de desbordamiento. No se puede indexar un array por encima de *n-1* o por debajo de 0 sin que Java provoque algún mensaje de error en compilación o ejecución.

Sin embargo, existen lenguajes, por ejemplo C++, que no realizan la comprobación de rango, dejando esta responsabilidad al programador.

4.3. Nivel de aplicación o uso de arrays en programas

Un array unidimensional es la estructura más idónea para modelar colecciones de elementos del mismo tipo de datos. Por ejemplo, listas de bodas, de precios, de números de teléfonos, de alumnos de un curso, etc.

Conviene recordar algunas operaciones absolutamente prohibidas en el uso de la estructura array y otras permitidas que ayudarán en la correcta utilización de arrays en programas de aplicación.

OPERACIONES PROHIBIDAS

Hay una serie de operaciones que no se realizan nunca con los arrays:

- Un array no puede ser destino de una asignación.

  Por ejemplo, las siguientes son todas expresiones incorrectas.

  int [] array= 0

  tabla = 0

  tabla = array //si ambos son objetos si es posible y significa que ambas referencias apuntarán al mismo contenido.

- No se puede leer o escribir de una vez el contenido de un array.

  Leer (tabla) o Leer (tabla[])

  Escribir (“El array es:”, tabla)

- Sobre un array no pueden realizarse operaciones aritméticas de una vez.

  tabla + array

En resumen, cualquier operación sobre arrays debe hacerse sobre elementos individuales de la estructura.

OPERACIONES FRECUENTES

Se detallan a continuación las operaciones con arrays más frecuentes.

- **Recorrido de un array**: un array se recorre con un índice que se mueve desde el primero al último elemento o desde el último al primero.

  En pseudocódigo se resolvería del siguiente modo:

  Para (i=0 ; mientras i<n ; i++) // del primero al último

  <Proceso>

Fin_Para

Para (i=n-1;mientras i>=0;i--) // del último al primero

<Proceso>

Fin_Para

- **Lectura y escritura**: un array puede recorrerse para leer o para escribir sus elementos.

  Ejemplo:

  Se desea guardar las notas de una asignatura de los alumnos de un grupo y después visualizar en pantalla los datos almacenados.

  Este ejercicio hace uso de un array de reales y de índice con significado.

```
//PruebaArray.java
import java.util.Scanner;
import java.io.*;
public class PruebaArray {
public static void main (String [] args) {
 final int MAX=10;
 int i;
 double notas[]= new double[MAX];
 Scanner teclado = new Scanner (System.in);
 /*Recorrido para leer datos de teclado y almacenar en el
array. Se diseñará como subprograma en los ejercicios de
aplicación.*/
 for(i=0;i<MAX;i++)
 {
    //Obtener los datos
    System.out.print('\n'+"Nota del alumno "+(i+1) +"  ");
    notas[i] = teclado.nextDouble();
 }
 /*Recorrido para escribir en pantalla los datos del array.
  Se diseñará como subprograma en los ejercicios de
 aplicación.*/
 for(i=0;i<notas.length;i++)
 {
  System.out.println("La nota del alumno "+(i+1)+"es:"
   +notas[i]);
 }
}
}//fin main
```

- **Inicialización:** en Java, cuando se crea el objeto y no se inicializa todos los elementos toman un valor inicial. Si se trata de tipos básicos, si son de números toman el valor cero, si de caracteres el valor '\0', o el valor **false**

si son del tipo booleano. Si se trata de referencias a objetos, el valor por defecto es **null.**

En Java se puede inicializar un array de tres formas diferentes:

1. A la vez que se declara según el siguiente formato:

   ```
   Tipo []nombreArray={lista de valores};
   ```

   Donde **lista de valores** son los valores asignados a cada una de las correspondientes casillas del array, separados por comas. Esta forma sólo se usa para arrays pequeños o bien para cadenas. **En Java se permite no dimensionar el array si se inicializa al declararlo**. Por ejemplo:

   ```
   int []array = {0, 1, 2, 3, 4};
   ```

   Por ejemplo,

   ```
   String []capitales=
   {"Avila","Burgos","León","Palencia","Salamanca",
      "Segovia","Soria","Valladolid","Zamora"};
   ```

   En este caso no hace falta que indiquemos numéricamente el número de elementos, sino que el tamaño se calculará dependiendo de los elementos que pongamos entre las llaves.

2. La segunda forma es mediante la asignación directa de elementos. En este caso el array habrá sido declarado con anterioridad:

   ```
   b[0] = "Avila";
   b[1] = "Burgos";
   b[2] = "León";
   ...
   ```

3. Recorriendo el array asignando a cada casilla un valor. Esta es la forma más usual de inicializar un array.

   Los valores a los que se suele inicializar un array son 0 o el índice si es de enteros, y si es de caracteres a blancos o a una cadena específica. Por ejemplo:

   Se pueden inicializar en un bucle **for** como resultado de alguna operación

   ```
   for(i=0; i<TAM; i++){
     numeros[i]=i*i+4;
   }
   ```

   El número de elementos del array nos lo proporciona su propiedad *length*. Escribimos de forma equivalente

   ```
   for(int i=0; i<numeros.length; i++){
     numeros[i]=i*i+4;
   }
   ```

VARIABLE DE INSTANCIA lenght

El objeto **Array** incluye una propiedad que es la longitud y que tiene el nombre **length**. La variable es pública y se puede acceder a ella directamente como un campo del objeto.

En el ejemplo siguiente la salida es 100:

```
public class ejemplo{
 public static void main(String[] args){
   int[] notas = new int[100];
   int cuantos = notas.length;
   System.out.println(cuantos);
 }
}
```

La longitud del array se establece cuando se crea, siendo fija a partir de entonces. La variable de instancia **length** es de tipo *final*, no puede ser modificada.

EJEMPLOS

Ejemplo1:

```
public class Ejemplo1{
    public static void main(String[] args){
      boolean[] miArray = new boolean[4];
      for( int i = 0; i< miArray.length;i++)
            System.out.print(miArray [i]+" ");
    }
}
```

La salida del programa es: `false false false false`

Ejemplo2:

```
public class Ejemplo2{
    public static void main(String[] args){
      double[]miArray = new double[10];
      for( int i = 0; i< miArray.length;i++)
          System.out.print(miArray [i]+" ");
    }
}
```

La salida del programa es: `0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0`

Ejemplo3:

```
public class Ejemplo3{
 public static void main(String[] args){
  double[] primos= {2.,3.,5.,7.,11.,13.,17.,19.};
  for(int i = 0;i<primos.length;i++) {
     primos[i]-=1;
     System.out.print(primos[i] +" ");}
 }
}
```

La salida del programa será: `1.0 2.0 4.0 6.0 10.0 12.0 16.0 18.0`

Ejemplo 4: Programa que muestra los parámetros que se pasan mediante la línea de comandos:

```
public  class  Eco{
 public  static  void  main(String  args[]){
  int  i;
  for (i=0; i<args.length; i++)
   System.out.println(args[i]);
 }
}
```

EJEMPLO ARRAY DE OBJETOS

Ejemplo 1: Suponiendo creada la clase *Rectangulo* para crear un array de objetos de la clase y usarlo, tendríamos que realizar los siguientes pasos:

- Declaración

```
Rectangulo[] rectangulos;
```

- Crear el array

```
rectangulos=new Rectangulo[3];
```

- Inicializar los elementos del array

```
rectangulos[0]=new Rectangulo(10,20,30,40);
rectangulos[1]=new Rectangulo(30,40);
rectangulos[2]=new Rectangulo(50,80);
```

O bien, en una sola línea

```
Rectangulo[] rectangulos={new Rectangulo(10,20,30,40),
 new Rectangulo(30,40), new Rectangulo(50,80)};
```

- Usar el array

Para calcular y mostrar el área de los rectángulos escribimos

```
for(int i=0; i<rectangulos.length; i++){
   System.out.println(rectangulos[i].calcularArea());
}
```

Ejemplo 2: si tengo una clase `Habitacion` con su constructor:

```
public Habitacion (int metros, int [ ] idVentanas) {
// ......
}
Habitacion salon = new Habitacion(15, new int[]{2,3,6});
```

COPIAS DE ARRAYS

La copia de los elementos de una Array se realiza tan frecuentemente que existe un método para ello en la clase *System*. Es el siguiente,

```
System.arraycopy(afuente,ifuente,adestino,idestino, numero)
```

Los argumentos: `afuente` es el array con los datos de origen, `adestino` el array dónde se copiarán los datos, `ifuente` el índice inicial para empezar a copiar elementos en el array original, `idestino` el índice inicial donde empiezan a copiarse los elementos en el destino, y `numero` el número de elementos que se han copiar.

Para copiar todos los elementos de un array a otro, por ejemplo los elementos de `a` al array `b`, la llamada al método sería:

```
System.arraycopy(a,0,b,0,a.length);
```

El array destino tiene que haberse creado con un tamaño suficiente para almacenar todos los elementos que se quieren copiar.

En el siguiente ejemplo se utiliza este método para copiar íntegramente un array, y se puede comprobar además la diferencia entre la copia de los elementos y la asignación de los objetos.

```
/**
*  Prueba para copiar arrays
*
* Última revisión: febrero, 2013
*/
import java.util.Scanner;
import java.io.*;

public class CopiaArray{

 public static void main(String[] args){

  int[] a = {12,14,16,18,20,22,24,26};
  int[] b = new int[a.length]; //Se crea con igual longitud

  System.arraycopy(a,0,b,0,a.length);//método de System

  System.out.print(" aaaa: ");
  for(int i=0;i<b.length;i++)
     System.out.print(a[i]+ " ");
  System.out.println();

  System.out.print(" bbbb: ");
  for(int i=0;i<b.length;i++)
     System.out.print(b[i]+ " ");
  System.out.println();
```

```
    a[1] = 3003;
    System.out.print(" aaaa cambiada: ");
    for(int i=0;i<b.length;i++)
       System.out.print(a[i]+ " ");
    System.out.println();

    System.out.print(" bbbb: ");
    for(int i=0;i<b.length;i++)
       System.out.print(b[i]+ " ");
    System.out.println();

    b = a; //como un array es un objeto y por tanto una
           //referencia esta asignación significa que
           //tenemos dos nombres para el mismo objeto.
    a[1] = 4004;

    System.out.print(" aaaa cambiada: ");
    for(int i=0;i<b.length;i++)
       System.out.print(a[i]+ " ");
    System.out.println();

    System.out.print(" b tras asignarle a: ");
    for(int i=0;i<b.length;i++) System.out.print(b[i]+ " ");
    }
}
```

y la salida es:

```
aaaa: 12 14 16 18 20 22 24 26
bbbb: 12 14 16 18 20 22 24 26
aaaa cambiada: 12 3003 16 18 20 22 24 26
bbbb: 12 14 16 18 20 22 24 26
aaaa cambiada: 12 4004 16 18 20 22 24 26
b tras asignarle a: 12 4004 16 18 20 22 24 26
```

CAMBIO DE TAMAÑO

Si un array se nos queda pequeño una solución sería crear uno nuevo de mayor dimensión y copiar los elementos. En el siguiente ejemplo se utiliza un método para aumentar de tamaño un array:

```
/**
*  Prueba para aumentar longitud de un array
*
* Última revisión: febrero, 2013
*/
import java.util.Scanner;
import java.io.*;

public class AumentaLongitud{

 public static int[] aumentaIntArray(int[] ant, int cant){
   int[] nue = new int[ant.length + cant];
```

```
    for(int i = 0;i< ant.length; i++)
      nue[i] = ant[i];
    return (nue);
   }
 public static void main(String[] args){
   int[]  a = {21,22,23};  int i;

   System.out.print("aaa: ");
   for(i=0;i<a.length;i++)
     System.out.print(a[i]+" ");
   System.out.println();

   a = aumentaIntArray(a,3);

   a[3] = 24; a[4] = 25; a[5] = 26;

   System.out.print("nueva aaa: ");
   for(i=0;i<a.length;i++)
     System.out.print(a[i]+" ");
   System.out.println();
  }
}
```

y la salida es:

```
aaa: 21 22 23
nueva aaa: 21 22 23 24 25 26
```

USANDO CLONE Y EQUALS

Con ***clone*** se puede duplicar un array, es decir, crear otro array con el mismo contenido. El método ***equals*** permite determinar si las variables hacen referencia al mismo objeto, en este caso, al mismo array. Por ejemplo :

```
/**
*  fuente : PruebaArryCE.java
*Prueba clone y equal en arrays
*
* Última revisión: febrero, 2013
*/
import java.util.Scanner;
import java.io.*;


class PruebaArryCE
{
  public static void main (String[] args){
   int[] arrayOr = {10,20,30,40,50};

// copia se inicializa con los mismos datos de arrayOr
// usando el método clone
   int[] copia = (int[])arrayOr.clone();
```

```
    if (arrayOr.equals(copia))
       System.out.println("Es el mismo array");
    else
       System.out.println("Son arrays distintos");

    copia = arrayOr; // ahora referenciaran al mismo array
    if (arrayOr.equals(copia))
       System.out.println("Es el mismo array");
    else
       System.out.println("Son arrys distintos");
       }
}
```

y la salida es:

```
Son arrays distintos
Es el mismo array
```

ÍNDICES CON SIGNIFICADO

En algunos algoritmos se da la circunstancia de que el índice de un array, además de señalar la posición de un elemento, puede tener un significado específico. En estos casos, se dice que el índice tiene un contenido semántico.

Por ejemplo, es normal asignar a los 30 alumnos de un curso ordenados alfabéticamente un código en la lista de 1 a 30. Se puede hacer coincidir el índice de un array, que guarda información sobre las notas de cada alumno, con su código. De esta forma, la casilla primera del array almacenaría la nota del alumno de código 1, que además, es el primero de la lista de clase, como puede observarse en el ejemplo `PruebaArray.java` del apartado **Lectura y escritura**. En estos casos, se dice que se usa un índice con significado.

ARRAYS PARALELOS

Son arrays por lo general de distintos tipos de datos, cuyos elementos correspondientes a órdenes iguales están semánticamente relacionados de alguna manera. Tienen la ventaja de poder ser procesados a la vez para acceder a la información relacionada.

Por ejemplo, suponga que se dispone de dos arrays en los que se almacenan los códigos (enteros) de platos de un restaurante y los precios (reales) de esos platos. Los arrays son de tipo de datos diferentes, pero la información que almacenan está relacionada. Cada código de plato se corresponde casilla a casilla con su precio correspondiente, por tanto, pueden ser procesados paralelamente.

Gráficamente, se puede expresar esa relación de la siguiente manera:

platos		precios
1001	----------	12,50
1005	----------	24.50
1010		9.25

-------	----------	-------
1050	----------	7.46
1055	----------	5.78

Si se estuviera interesado en disponer de un listado en pantalla con la información de ambos arrays, en la que se visualizara código y precio de cada plato, se pueden procesar a la vez utilizando un único bucle para recorrerlos.

```
.....................
for (i = 0; i < platos.length; i++)
{
   System.out.println ("El plato de código   " +platos[i]+
   "cuesta "+precios[i]+"€");
}
.......................
```

La salida en pantalla de este trozo de código sería:

```
El plato de código    1001  cuesta  12.50 €
El plato de código    1005  cuesta  24.50 €
El plato de código    1010  cuesta  9.25 €
                      ...
                      ...
El plato de código    1050  cuesta  7.46 €
El plato de código    1055  cuesta  5.78 €
```

5. PASO DE ARRAYS A FUNCIONES

En Java un array es considerado un objeto por lo que siempre es pasado "por referencia". Recordad que realmente su comportamiento es como el paso por dirección de C/C++, es decir, paso por valor de una dirección.

Observa en el siguiente ejemplo como se hace el paso de parámetro cuando este es un array, así como la llamada a las funciones y como se devuelve el array asociado al nombre, es decir, mediante la sentencia *return*.

Ejemplo: Se trata del mismo ejemplo del apartado **Lectura y escritura** pero usando funciones y procedimientos.

```
/**
*  Program name: PasoParArrays.java
*
**/
import java.util.Scanner;
import java.io.*;

public class PasoParArrays {
 public static double[] cargarArray (int tam){
  double notas[]= new double[tam];
  Scanner teclado = new Scanner (System.in);
  for(int i=0;i<tam;i++)
```

```
        {
          //Obtener los datos
          System.out.print('\n'+"Nota del alumno "+i +"  ");
          notas[i] =teclado.nextDouble();

        }
        return notas;
    }//fin cargarArray

    public static void imprimirArray(double notasM[])
    {
    for(int i=0;i< notasM.length;i++)
    {
          System.out.println("La nota del alumno   " +(i+1)+"
            es: "+notasM[i]);
    }
    }//fin imprimirArray

    public static void main (String [] args) {
    final int MAX=10, tam =10;
    int i;
    double notasM[]= new double[MAX];

    //Llamada a función para cargar el array.
    //Observa que solo necesita el tamaño de dicho array
      notasM= cargarArray(tam);

    //Llamada a función para imprimir el array.
    // como puedes comprobar tan solo necesita el array
       imprimirArray(notasM);
    }
}
```

7. ARRAYS BIDIMENSIONALES

En los lenguajes de propósito general también están implementados los arrays de dos dimensiones, conocidos también como **matrices**. Se estudiarán atendiendo a la perspectiva en tres niveles.

7.1. Nivel lógico o abstracto

La mayor parte de lo expresado acerca de la visión abstracta de un array de una dimensión, se aplica también a la visión lógica de un array de dos dimensiones. La definición lógica de array bidimensional puede hacerse desde dos enfoques diferentes:

A. **Enfoque recursivo:** desde este punto de vista, se puede decir que un array bidimensional es un array de una dimensión en el que cada elemento es a su vez un array de una dimensión.

B. **Enfoque directo:** según este enfoque, un array bidimensional es una estructura de datos formada por una colección finita de elementos homogéneos, ordenado cada uno de ellos en dos dimensiones y referenciados bajo un nombre común.

La **función de acceso** a los elementos de la estructura es independiente del enfoque utilizado para la definición. El mecanismo para acceder es directo a un determinado componente. Es decir, no hay necesidad de "visitar" a los que le preceden para llegar a un elemento concreto. Cada elemento está ordenado en dos dimensiones, por tanto, el acceso se realiza mediante el nombre del array y usando un par de índices, que indican a que elemento de la colección se desea acceder. El primer índice referencia la posición del elemento en la primera dimensión, llamada *fila*, y el segundo índice señala la posición del elemento en la segunda dimensión, llamada *columna*.

La sintaxis y la semántica del lenguaje de programación en particular especificarán la función de acceso.

SINTAXIS DE LA FUNCIÓN DE ACCESO:

Cada elemento de un array de dos dimensiones puede accederse directamente mediante la siguiente expresión:

nombreArray [expresiónIndice1][expresiónIndice2]

nombreArray: **e**s un indicador que representa el nombre asociado a la colección de elementos de la estructura.

expresiónIndice1: es un valor comprendido entre 0 y *n-1*, siendo *n* el número de filas de la estructura.

expresiónIndice2: Es un valor comprendido entre 0 y *m-1*, siendo *m* el número de columnas de la estructura.

La cantidad de elementos del array vendrá dada por el producto de filas por columnas.

SEMÁNTICA DE LA FUNCIÓN DE ACCESO

La expresión de la función de acceso indica la localización del elemento asociado con la expresión de *índice1* e *índice2*, en la colección de elementos cuyo nombre es *nombreArray*. Un elemento se refiere con el nombre del array y un valor para cada uno de los índices entre corchetes.

La función de acceso, como en el array de una dimensión, se usa de dos formas:

1. Para especificar un lugar donde ha de almacenarse un valor.

   Ejemplo: Suponiendo declarado un array que guarda en la primera columna la temperatura máxima y en la segunda la mínima de cada día de un mes.

```
maxMin[15][1] = 10 /*maxMin es el nombre de la colección
                     y 15, 1 es la referencia al elemento
```

```
                        16 de la segunda columna del array
                        para almacenar el entero 10 */
Leer(maxMin[15][1]) /*Se almacena en el elemento 16 el
                        valor leído por teclado */

maxMin[i][j]= 0     /*[i][j] es la referencia del
                        elemento j-ésimo de la fila i-ésima
                        de la estructura */
```

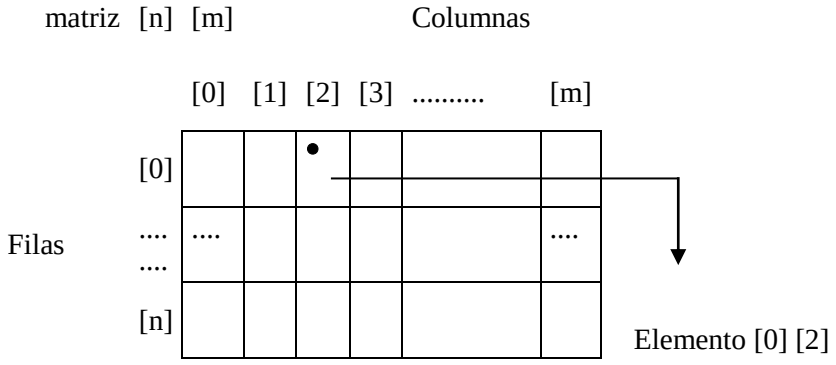
2. Para especificar o referenciar un lugar de donde ha de extraerse un valor.

```
var = maxMin[15][1] /*Asigna el valor del elemento 15,1
                    del grupo a la variable var */

Escribir ("Temperaturas del día --- Max: ---  Min: ---",
          i+1, maxMin [i][0], maxMin [i][1])
```

REPRESENTACIÓN GRÁFICA

Un array bidimensional es una estructura natural para almacenar información que se puede representar de manera lógica como una **matriz,** visualizándose de forma abstracta como una rejilla con filas y columnas, de manera que cada elemento se referencia por el número de fila y columna donde se encuentra, tal como aparece en la figura siguiente.



7.2. Nivel de implementación

Recuerde que, para implementar cualquier estructura de datos, han de realizarse dos tareas, en primer lugar, reservar memoria para la estructura y en segundo, codificar sus funciones de acceso. Como esta también es una estructura de datos incorporada, será el compilador quién realice estas funciones.

DECLARACIÓN E INICIALIZACIÓN

La reserva de memoria para un array de cualquier dimensión se realiza a través de la sentencia de declaración. Con ella, se informa al compilador de cuántas filas y

cuántas columnas se requieren para almacenar datos en la estructura, así como, el nombre de la estructura y el tipo de datos de sus elementos.

Formato de declaración en Java de variables de tipo Array de dos dimensiones:

tipo nombreArray [][];

o bien,

tipo [][]NombreArray; //más usado

tipo: Puede ser cualquier tipo de datos básico o definido por el usuario.

nombreArray: Es el identificador que referencia la estructura completa.

CREACIÓN

Como los de una dimensión se crean con new, al mismo tiempo que se declaran

tipo [][] nombreArray = new tipo[filas][columnas];

o cuando necesitemos una vez creados

nombreArray = new tipo[filas][columnas];

filas: Es la cantidad de filas de la colección, debe evaluarse como una constante entera positiva.

columnas: Refiere la cantidad de columnas de la colección, también constante entera y positiva.

INICIALIZACIÓN

Como los de una dimensión en Java se puede inicializar un array de dos dimensiones de tres formas diferentes:

1. A la vez que se declara según el siguiente formato:

```
tipo[][]nombreArray={{lista valores fila1}
                     {.....................}
                     {lista de valores fila n}};
```

Donde **lista de valores** son los valores asignados a cada una de las correspondientes filas del array, separados por comas.

Por ejemplo:

```
int [][]matriz = {{1,2}, //Primera fila
                  {3,4},  //Segunda fila
                  {5,6}}; //Tercera fila
```

En este caso no hace falta que indiquemos numéricamente el número de filas y columnas, sino que el tamaño se calculará dependiendo de los elementos que pongamos entre las llaves.

2. La segunda forma es mediante la asignación directa de elementos. En este caso el array habrá sido declarado con anterioridad:

```
matriz[0][0] = 1;
matriz[0][1] = 2;
matriz[1][0] = 3;
matriz[1][1] = 4;
matriz[2][0] = 5;
matriz[2][1] = 6;
```

3. Recorriendo el array asignando a cada casilla un valor. Esta es la forma más usual de inicializar un array.

   Por ejemplo: Se pueden inicializar usando dos bucles **for** como resultado de alguna operación

```
for(i=0; i<numeroFilas; i++){
  for(j=0; j<numeroColumnas; j++)
     matriz[i][j]=i*j+1;
}
```

   Como vimos anteriormente no necesitamos recordar el número de elementos de un array, en este caso podemos conocer el número de filas y el número de columnas de la matriz mediante *length*.

```
for(int i=0; i<matriz.length; i++){
  for(j=0; j<matriz[0].length; j++)
     matriz[i][j]=i*j+1;
}
```

VARIABLE DE INSTANCIA lenght

Para obtener el número de filas del array bidimensional podemos usar la propiedad *length* de los arrays, de la siguiente manera:

```
int filas = matriz.length;
```

Para saber el número de columnas sería de la siguiente forma :

```
int columnas = matriz[0].length;
```

También podemos clonar un array de dos dimensiones, es decir, crear una matriz nueva a partir de otra matriz creada con anterioridad, siguiendo esta sintaxis:

```
String[][] nuevaMatriz = matriz.clone();
```

CARACTERÍSTICAS DE LOS ARRAYS DE DOS DIMENSIONES

Las características de un array de dos dimensiones son las mismas que se han descrito anteriormente para el array de una dimensión, salvo que ahora se crean, además, una cota superior y otra inferior por cada una de las dimensiones.

FUNCIONAMIENTO INTERNO

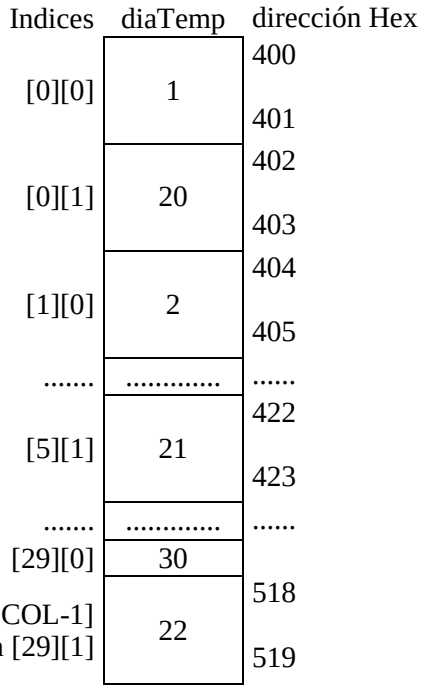
La función de acceso utiliza la información de la declaración para determinar la posición de un elemento. Se entiende que cada casilla ocupa la cantidad de memoria necesaria para almacenar un valor del tipo de dato declarado.

Aunque el array se visualiza como una matriz de filas por columnas, en Java se guarda en memoria con los elementos de cada fila en secuencia. Se dice que se almacena en orden principal de filas, es decir, en primer lugar los elementos de la primera fila, a continuación los elementos de la segunda, después los de la tercera y, así sucesivamente, hasta los elementos de la última fila. Esto es una característica del lenguaje, por ejemplo, **Pascal** almacena los arrays de dos dimensiones en orden principal de columnas, es decir, los elementos se almacenan en orden, los de una columna a continuación de los de otra. Esto no afecta al programador puesto que el mecanismo de acceso le es transparente. Sin embargo, habrá de tener en cuenta la forma en que el compilador almacena los elementos si se desea acceder a ellos a través de punteros que manejan directamente posiciones de memoria.

Por ejemplo, el siguiente array almacena en la primera columna los 30 días de un mes y en la segunda, las temperaturas máximas de cada día.

short [][]diaTemp = new short [FILAS] [COLUMNAS] , ocupa 2 bytes por elemento

Si la dirección de base es 400 H, la representación en memoria seria la siguiente:



Para hallar el tamaño en bytes de un array de dos dimensiones habrá que aplicar la siguiente fórmula:

TamañoBytes = Filas * Columnas * TamañoTipo

La función de acceso para hallar el lugar que ocupa un elemento cualquiera del array, siendo *FIL* y *COL* la casilla correspondiente al elemento al que se desea acceder, será la siguiente:

Dir NombreArray[FIL][COL] = Base+(Columnas*FIL+COL)*TamañoTipo

Siguiendo el ejemplo, el elemento [5][1] ocupará las posiciones de memoria 422 y 423, y el elemento [29][1], último de la estructura, estará almacenado en las direcciones 518 y 519, como puede observarse aplicando la fórmula anterior.

Dirección de comienzo de diaTemp [5] [1] = 400 + (2 * 5 + 1) * 2 = 422

Dirección de comienzo de diaTemp [29] [1] = 400 + (2 * 29 + 1) * 2 = 518

En definitiva, según las características asociadas a un array y utilizando la implementación de la función de acceso, el compilador de Java accede a cualquier elemento de un array de dos dimensiones.

7.3. Nivel de aplicación o de uso

El uso más generalizado de los arrays de dos dimensiones en los problemas de gestión, es para guardar distinta información sobre un mismo objeto, siempre que dicha información sea del mismo tipo de datos, por ejemplo, los códigos y las notas de los módulos de los alumnos de primero del ciclo, suponiendo que código y notas son del mismo tipo de datos.

En otro entorno, distinto al de gestión, los arrays bidimensionales son útiles para modelar objetos bidimensionales, tales como, tableros de ajedrez, cartones de bingo, laberintos o cualquier otro juego que se desarrolle sobre tableros de dos dimensiones. También sirven para modelar matrices matemáticas, planos, etc.

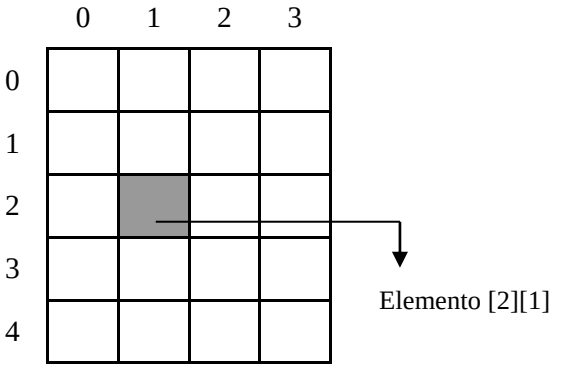
Independientemente de lo que represente un array bidimensional en un programa, es decir, la semántica de los valores que trate, pueden realizarse con ellos las siguientes operaciones:

1. Acceso aleatorio o directo a un elemento en particular.
2. Acceso de forma sistemática a cada elemento por filas o por columnas, o sea, recorrido del array.
3. Procesar una fila.
4. Procesar una columna.

ACCESO DIRECTO A UN ELEMENTO

A veces se puede estar interesado en procesar una celda dada de un array de dos dimensiones. Por ejemplo, proceso del elemento correspondiente a la casilla 2,1, a la que se accedería del siguiente modo:

< Procesar tabla [2][1]>



RECORRIDO

El recorrido de un array de dos dimensiones, al igual que los unidimensionales, se realiza para leer, escribir, inicializar y, en general, para procesar todos los elementos del array. Se puede acceder a cada uno de los elementos realizando el recorrido por filas o por columnas. El hecho de realizar el recorrido en un sentido o en otro, no influye en absoluto sobre el proceso que se debe realizar con los elemento, sólo atañe a la forma en que esos elementos son accedidos en la estructura de datos.

Es habitual el recorrido para inicializar arrays lo suficientemente grandes como para que no sea eficaz inicializarlos a la vez que se declaran.

A continuación, se muestran los bucles que determinan el recorrido, bien sea por filas o por columnas, de todos los elementos del array de dos dimensiones.

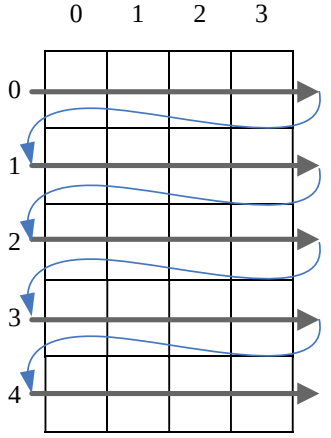
- **por filas**:

```
Para (i=0, mientras i<FIL, i++)
   /* i es el índice de filas */
    Para (j=0, mientras j<COL, j++)
       /* j es el índice de columnas */

            < Procesar tabla [i][j] >

            /* tabla [i][j] = valor Si
             se desea inicializar el array */

   Fin_Para
Fin_Para
```



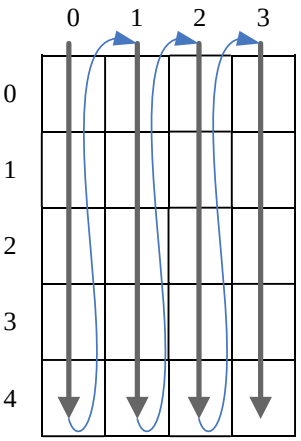
El *valor* al que se inicializa el array suele ser 0 para enteros, blanco para caracteres o cualquier otro que convenga en el contexto del programa.

- **por columnas**:

```
Para (j=0, mientras j<COL, i++)
   Para (i=0, mientras i<FIL, i++)

         < Procesar tabla [i][j] >

   Fin_Para
Fin_Para
```



Advierta que usando ambas formas de recorrido, se ha accedido para su proceso al mismo elemento, *tabla [i][j]*.

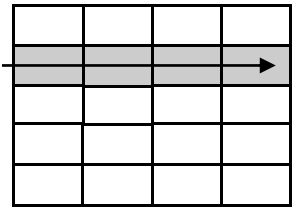
PROCESAR UNA FILA

Resulta relativamente frecuente que la implementación de un algoritmo requiera el proceso de una sola fila, porque semánticamente así lo exija el contexto del programa que se está desarrollando. En este caso, es posible acceder a una fila en concreto de la forma que se detalla a continuación.

```
Para (j=0, mientras j<COL, j++)

   < Procesar tabla [filaConcreta][j] >

Fin_Para
```



En el gráfico superior se está procesando la fila 1 en particular.

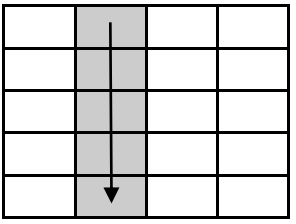
PROCESAR UNA COLUMNA

Puede que el ejercicio que se está diseñando requiera el proceso de una columna en particular, en cuyo caso se procedería del siguiente modo:

```
Para (i=0, mientras i<FIL, i++)

   < Procesar Tabla [i][columnaConcreta]

Fin_Para
```



En el gráfico superior se está procesando particularmente la columna 1.

EJERCICIO PRÁCTICO DE APLICACIÓN

Se desea realizar el tratamiento de las temperaturas máximas de los días de un mes para calcular la temperatura media de cada uno de ellos y la del año.

8. PASO DE ARRAYS DE DOS DIMENSIONES A FUNCIONES

En Java los arrays de dos dimensiones pueden ser pasados a funciones y devueltos por ellas como cualquier otro tipo de objeto.

Por ejemplo, siendo las siguientes funcionalidades

```
void Proced (int [][] matriz){...}
int [][] Func1 (int filas, int columnas){...}
```

La llamada sigue realizándose, como con los arrays de una dimensión, enviando el nombre del array. Siendo la matrices *matriz* y *matrizA*, la llamada se haría del siguiente modo:

```
Proced (matrizA);
matriz = Func (fil, column);
```

9. ARRAYS N-DIMENSIONALES

Por regla general, en todos los lenguajes donde están implementados los arrays de una y dos dimensiones, también lo están los arrays de *n* dimensiones o multidimensionales, conocidos también como **poliedros**. Generalmente, no se suele poner límite a la cantidad de dimensiones de un array, salvo las que impone la arquitectura física del ordenador, es decir, la capacidad de la memoria, ya que se trata de estructuras estáticas de datos que consumen mucha memoria.

Las estructuras de datos arrays de *n* dimensiones, conceptualmente, se definen como los arrays uni y bi-dimensionales, salvo que los elementos están ordenados en *n* dimensiones.

Formato de declaración:

tipo nombreArray [][][][][]……[];

El acceso a elementos de esta estructura se realiza mediante el uso de tantos índices como dimensiones tenga la estructura.

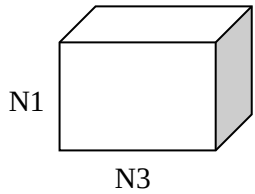
No es aconsejable usar este tipo de estructuras por la complejidad que conllevan y por la gran cantidad de memoria que consumen. Siendo N_i el tamaño de cada dimensión, la cantidad de memoria viene dada por la fórmula:

Nº de bytes = N1*N2*N3*N4*…….*Nx * bytes del tipo

Si algún algoritmo requiere una estructura similar, es el momento de estudiar el uso de una estructura de datos distinta que resuelva el problema, ocupe menos memoria, y haga los algoritmos más sencillos, por ejemplo, los archivos.

La visión abstracta de la estructura de tres dimensiones puede representarse gráficamente como un cubo lleno de celdillas. Para estructuras de más dimensiones es difícil, por no decir imposible, representarlas gráficamente.

- N1 representa las filas
- N2 representa las columnas
- N3 representa la anchura



Un ejemplo de uso de esta estructura puede ser la implementación del famoso cubo de Rubik.

EJERCICIO PRÁCTICO DE APLICACIÓN

Acaba de celebrarse elecciones en la ciudad. Hay que realizar un recuento y análisis de los votos de los cincos candidatos por cada uno de los cuatro distritos que existen en la ciudad. Se ha de reflejar, además, cuántos votos totales recibió cada candidato y cuántos votos totales hubo en cada distrito, así como el total de votantes del Ayuntamiento.

Estructura de datos que se usará: un array de dos dimensiones donde se almacenarán los votos de cada candidato (Filas) en cada distrito (Columnas). Se añadirá una fila más para acumular los totales por distrito y una columna más para acumular los totales por candidato. Los nombres de los candidatos se guardarán en un array de cadenas que se inicializa al declararlo.

ANEXO:

Arrays de caracteres.

Arrays de cadenas.

Clases arrays en Java

Anotaciones

NOTA:

En el lenguaje de programación Java, un array multidimensional es una matriz cuyos componentes son a su vez arrays. Esto es una diferencia importante con las matrices en C o Fortran. Una consecuencia de esto es que las filas pueden variar en longitud, como se muestra en el siguiente ejemplo:

```
class MultiDimArrayDemo {

    public static void main(String[] args){
        String[][] names = {
            {"Mr. ","Mrs. ","Ms. "},
            {"Smith","Jones"}};

        // Mr. Smith
        System.out.println(names[0][0] +
                           names[1][0]);
        // Ms. Jones
        System.out.println(names[0][2] +
                           names[1][1]);
    }
}
```

La salida de este programa es:

```
Mr. Smith
Ms. Jones
```

COPYING ARRAYS

The `System` class has an `arraycopy` method that you can use to efficiently copy data from one array into another:

```
public static void arraycopy(Object src, int srcPos,
                             Object dest, int destPos, int length)
```

The two `Object` arguments specify the array to copy *from* and the array to copy *to* . The three `int` arguments specify the starting position in the source array, the starting position in the destination array, and the number of array elements to copy.

The following program, `ArrayCopyDemo` , declares an array of `char` elements, spelling the word "decaffeinated." It uses the `System.arraycopy` method to copy a subsequence of array components into a second array:

```
class ArrayCopyDemo {
    public static void main(String[] args) {
        char[] copyFrom = { 'd', 'e', 'c', 'a', 'f', 'f', 'e',
                            'i', 'n', 'a', 't', 'e', 'd' };
        char[] copyTo = new char[7];

        System.arraycopy(copyFrom, 2, copyTo, 0, 7);
        System.out.println(new String(copyTo));
    }
}
```

The output from this program is:

```
caffein
```

ARRAY MANIPULATIONS

Arrays are a powerful and useful concept used in programming. Java SE provides methods to perform some of the most common manipulations related to arrays. For instance, the `ArrayCopyDemo` example uses the `arraycopy` method of the `System` class instead of manually iterating through the elements of the source array and placing each one into the destination array. This is performed behind the scenes, enabling the developer to use just one line of code to call the method.

For your convenience, Java SE provides several methods for performing array manipulations (common tasks, such as copying, sorting and searching arrays) in the `java.util.Arrays` class. For instance, the previous example can be modified to use the `copyOfRange` method of the `java.util.Arrays` class, as you can see in the `ArrayCopyOfDemo` example. The difference is that using the `copyOfRange` method does not require you to create the destination array before calling the method, because the destination array is returned by the method:

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

        char[] copyFrom = {'d', 'e', 'c', 'a', 'f', 'f', 'e',
            'i', 'n', 'a', 't', 'e', 'd'};

        char[] copyTo = java.util.Arrays.copyOfRange(copyFrom, 2, 9);

        System.out.println(new String(copyTo));
    }
}
```

As you can see, the output from this program is the same (`caffein`), although it requires fewer lines of code. Note that the second parameter of the `copyOfRange` method is the initial index of the range to be copied, inclusively, while the third parameter is the final index of the range to be copied, *exclusively* . In this example, the range to be copied does not include the array element at index 9 (which contains the character `a`).

Some other useful operations provided by methods in the `java.util.Arrays` class, are:

- Searching an array for a specific value to get the index at which it is placed (the `binarySearch` method).
- Comparing two arrays to determine if they are equal or not (the `equals` method).

- Filling an array to place a specific value at each index (the `fill` method).
- Sorting an array into ascending order. This can be done either sequentially, using the `sort` method, or concurrently, using the `parallelSort` method introduced in Java SE 8. Parallel sorting of large arrays on multiprocessor systems is faster than sequential array sorting.