Arreglos Multidimensionales

Programación Orientada a Objetos (CC2008) - 2026

Arreglos Multidimensionales

Semestre 02, 2026

El dato que tiene dos coordenadas

Un arreglo de una dimensión resuelve bien las listas: notas, nombres, precios. Cada elemento se localiza con un solo número.

Pero hay datos que no son una lista. Un tablero de tres en raya, la pantalla de un teléfono, una hoja de cálculo: en todos, una casilla se localiza con dos números, no con uno.

```
         col 0   col 1   col 2
       ┌───────┬───────┬───────┐
fila 0 │   X   │   O   │   X   │
       ├───────┼───────┼───────┤
fila 1 │   O   │   X   │   .   │
       ├───────┼───────┼───────┤
fila 2 │   .   │   .   │   O   │
       └───────┴───────┴───────┘
```

Se podría aplanar el tablero en un arreglo de nueve casillas y calcular la posición con `fila * 3 + columna`. Funciona, pero obliga a repetir esa cuenta en cada acceso, basta equivocarse una vez para corromper el tablero, y el código deja de

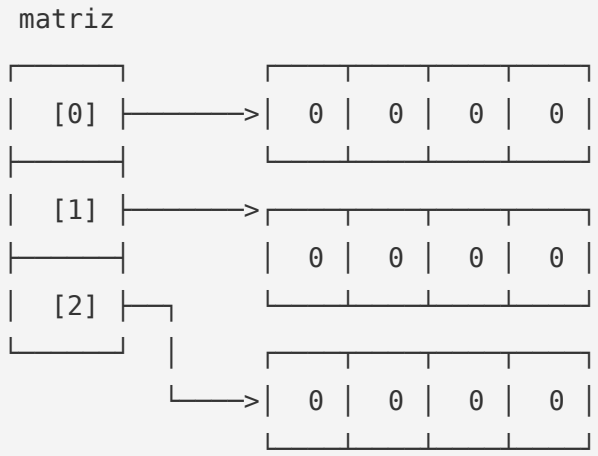
parecerse al problema. El arreglo multidimensional evita esa traducción.

Un arreglo cuyas casillas son arreglos

En Java no existe una estructura especial de matriz. Lo que existe es un arreglo cuyos elementos son, a su vez, arreglos.

```
int[][] matriz = new int[3][4];
```

Esa línea crea cuatro objetos en memoria: un arreglo externo de tres casillas y tres arreglos internos de cuatro casillas cada uno.



Las casillas del arreglo externo no guardan números: guardan referencias. Es el mismo mecanismo de los arreglos de objetos que ya vimos, solo que aquí los objetos apuntados son arreglos. Cada fila es un arreglo completo e independiente, con su propia longitud.

Declaración y creación

Hay tres formas de crear una matriz, y se eligen según lo que se sepa al momento de escribir el código.

Con tamaño conocido

```
int[][] matriz = new int[3][4];
```

Reserva la estructura completa y llena todo con el valor por defecto del tipo: `0` para los numéricos, `false` para `boolean`, `null` para las referencias.

Con valores literales

```
int[][] notas = {
    {80, 75, 90},
    {88, 92, 79},
    {65, 70, 72}
};
```

El compilador deduce que son tres filas de tres columnas. Es la forma más clara cuando los datos se conocen de antemano.

Por partes

```
int[][] matriz = new int[3][];

matriz[0] = new int[4];
matriz[1] = new int[2];
matriz[2] = new int[7];
```

Solo se reserva el arreglo externo y cada fila se crea después. Es lo que permite que las filas tengan longitudes distintas. Entre las dos instrucciones las filas valen `null`, y usarlas antes de crearlas lanza una excepción de puntero nulo.

Los dos índices

El acceso a una casilla usa dos corchetes, y cada uno significa una cosa distinta.

```
matriz[1][2] = 45;
```

- El primer índice elige la fila, es decir cuál de los arreglos internos se va a usar.
- El segundo índice elige la columna, es decir la posición dentro de ese arreglo.

Leerlo en dos tiempos ayuda: `matriz[1]` es un arreglo completo, y `matriz[1][2]` es la casilla número dos de ese arreglo.

```
int[] segundaFila = matriz[1];
int valor = segundaFila[2];
```

Como es una referencia y no una copia, modificar `segundaFila` modifica la matriz original.

El tamaño de cada dimensión

Como cada fila es un arreglo independiente, cada una tiene su propia propiedad `length`.

```
matriz.length       // cantidad de filas
matriz[0].length    // columnas de la primera fila
matriz[2].length    // columnas de la tercera fila
```

Escribir `matriz[0].length` para todas las filas solo es correcto si la matriz es rectangular. Si las filas pueden tener largos distintos, hay que consultar el `length` de cada una.

Recorridos

El orden del recorrido

Recorrer una matriz completa pide dos ciclos. El externo avanza por las filas y decide en cuál se está trabajando; el interno recorre todas las columnas de esa fila antes de que el externo avance.

El resultado es el orden de lectura de un texto: se termina una fila completa antes de bajar a la siguiente.

El cuerpo del ciclo interno corre una vez por cada casilla. En una matriz de 3 por 4 son 12 ejecuciones, y el costo crece rápido: duplicar filas y columnas cuadruplica el trabajo. Una imagen de 1000 por 1000 son un millón de visitas por cada filtro.

El límite del ciclo interno se consulta sobre la fila actual y no sobre la primera, para que el recorrido funcione también con filas de largos distintos.

Recorrido por columnas

Intercambiar el papel de los dos ciclos cambia el orden de visita sin cambiar la estructura ni un solo dato.

```
orden de visita, columna por columna


┌────┬────┬────┬────┐
│  1 │  4 │  7 │ 10 │
├────┼────┼────┼────┤
│  2 │  5 │  8 │ 11 │
├────┼────┼────┼────┤
│  3 │  6 │  9 │ 12 │
└────┴────┴────┴────┘
```

Ahora el externo avanza por columnas y el interno por filas: la columna queda fija mientras se baja por ella. Sirve para agregar por columna, como el promedio de cada examen en vez del promedio de cada alumno. Este recorrido asume que la matriz es rectangular.

Los patrones de índices

No todo recorrido usa dos ciclos. Cuando los índices siguen un patrón, basta con una sola variable.

- Fila: el primer índice queda fijo y el segundo avanza.
- Columna: el segundo queda fijo y el primero avanza.
- Diagonal principal: los dos avanzan juntos, así que siempre coinciden.
- Diagonal inversa: uno crece mientras el otro decrece, y suman siempre lo mismo.

La matriz siempre es la misma. Lo que cambia entre calcular un promedio por alumno, uno por examen o revisar una diagonal es únicamente el patrón con que se mueven los dos índices. Antes de escribir un ciclo conviene dibujar qué casillas hay que tocar y en qué orden: reconocido el patrón, escribir el ciclo es mecánico.

Arreglos irregulares

Nada obliga a que todas las filas midan lo mismo. Cuando difieren, el arreglo se llama irregular o dentado.

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

triangulo[0] = new int[1];
triangulo[1] = new int[2];
triangulo[2] = new int[3];
triangulo[3] = new int[4];
```

Es posible porque cada fila es un objeto independiente: el arreglo externo solo guarda referencias, y no le importa a qué tamaño apuntan.

También se pueden declarar con literales:

```
int[][] pascal = {
    {1},
    {1, 1},
    {1, 2, 1},
    {1, 3, 3, 1}
};
```

En un arreglo irregular es obligatorio usar `matriz[i].length` en el ciclo interno. Escribir `matriz[0].length` provoca un error de índice fuera de rango en cuanto

una fila posterior sea más corta que la primera. Como funciona igual en matrices rectangulares, conviene escribirlo siempre así.

Cuándo conviene

- Estructuras naturalmente triangulares, como el triángulo de Pascal.
- Listas de longitud variable por categoría: los alumnos inscritos en cada sección.
- Ahorro de memoria cuando la mayoría de las filas son mucho más cortas que la más larga.

Tres dimensiones

Agregar un índice más agrega una dimensión más. Una imagen a color es el ejemplo típico: cada píxel ya no es un número sino tres, uno por canal.

```
int[][][] imagen = new int[alto][ancho][3];

imagen[y][x][0] = 255;   // canal rojo
imagen[y][x][1] = 128;   // canal verde
imagen[y][x][2] = 0;     // canal azul
```

La estructura se lee de afuera hacia adentro: un arreglo de filas, donde cada fila es un arreglo de píxeles, donde cada píxel es un arreglo de tres canales.

Recorrerla completa pide tres ciclos anidados: el más externo baja por las filas, el intermedio avanza por las columnas hasta llegar a un píxel concreto, y el interno visita el rojo, el verde y el azul.

Otros usos de tres dimensiones son un volumen de vóxeles, una serie de tableros a lo largo del tiempo, o un conjunto de imágenes apiladas.

El caso general de n dimensiones

La sintaxis no tiene un límite práctico: cada par de corchetes agrega una dimensión.

```
int[][][][] tensor = new int[2][3][4][5];
```

Vale la pena entender el costo antes de usarlo. El número de casillas es el producto de todas las dimensiones, así que crece muy rápido.

- `new int[100][100]` reserva 10 mil casillas.
- `new int[100][100][100]` reserva 1 millón.
- `new int[100][100][100][100]` reserva 100 millones, unos 400 MB.

En la práctica es raro pasar de tres dimensiones. Cuando el problema parece pedir más, casi siempre conviene modelarlo con clases: en `datos[2][7][3][1]` ningún índice dice qué significa, mientras que en `sucursales[2].ventas[7]` cada nivel tiene nombre y el compilador ayuda si se escribe mal.

Caso de uso: matriz de adyacencia

Una red de conexiones se guarda en una matriz cuadrada: tantas filas y columnas como nodos tenga la red. Un uno significa que los dos nodos están conectados.

```
      0   1   2   3   4
    ┌───┬───┬───┬───┬───┐
0   │ 0 │ 1 │ 0 │ 1 │ 0 │
    ├───┼───┼───┼───┼───┤
1   │ 1 │ 0 │ 1 │ 0 │ 0 │
    ├───┼───┼───┼───┼───┤
```

```
    2 | 0 | 1 | 0 | 1 | 0 |
      ├───┼───┼───┼───┼───┤
    3 | 1 | 0 | 1 | 0 | 1 |
      ├───┼───┼───┼───┼───┤
    4 | 0 | 0 | 0 | 1 | 0 |
      └───┴───┴───┴───┴───┘
```

El dibujo de la red y la matriz contienen exactamente la misma información.

Una fila es un nodo

La fila tres reúne todas las conexiones del nodo tres: se lee de corrido, sin recorrer el resto de la matriz. Sumar la fila da el número de conexiones del nodo, que en un grafo se llama su grado. La columna tres dice lo mismo desde el otro lado: quién apunta hacia ese nodo.

La simetría

Si la relación es mutua, cada conexión aparece dos veces: en la casilla y en su reflejo respecto a la diagonal. La casilla de la fila cero y columna tres, y la de la fila tres y columna cero, guardan el mismo dato.

La diagonal marca las casillas donde un nodo se compara consigo mismo, y normalmente vale cero. Cuando la relación no es mutua, como seguir a alguien en una red social, la matriz deja de ser simétrica.

Preguntas que responde la matriz

Cada pregunta sobre la red se convierte en un recorrido distinto sobre la misma matriz. Cambia el patrón de índices, no los datos.

Conexión directa entre dos nodos

Se responde leyendo una sola casilla. La de la fila cero y la columna tres vale uno, así que el 0 y el 3 están conectados. La de la fila cero y la columna dos vale cero, así que entre el 0 y el 2 no hay conexión directa.

Es la operación más barata de la matriz: una sola lectura, sin importar cuántos nodos tenga la red.

Grado de un nodo

El grado es cuántas conexiones tiene un nodo, y se obtiene sumando toda su fila. La fila tres suma `1 + 0 + 1 + 0 + 1 = 3`, que es el número de líneas que salen del nodo 3 en el dibujo.

Es un recorrido de una sola dimensión: se fija la fila y se avanza por las columnas.

El nodo más conectado

Calcular el grado de todos y quedarse con el mayor. Ahora sí hacen falta los dos índices: el ciclo externo recorre las filas y el interno suma cada una.

```
fila 0 -> 2      fila 3 -> 3   <- el mayor
fila 1 -> 2      fila 4 -> 1
fila 2 -> 2
```

En una red social esto identifica a la cuenta más influyente.

Un nodo aislado

Un nodo sin ninguna conexión deja su fila entera en ceros, y basta recorrerla verificando que no aparezca ningún uno. Si se elimina la conexión entre el 3 y el 4, la fila 4 queda en ceros y el nodo 4 se desconecta de la red.

El nodo 4 tenía grado uno, así que era el más frágil: perder una sola conexión lo aísla por completo.

Amigos en común

Comparar dos filas y contar en qué columnas ambas tienen un uno. Las filas 0 y 2 coinciden en las columnas uno y tres, así que los nodos 0 y 2 comparten al 1 y al 3 como vecinos.

El recorrido avanza por las columnas comparando dos filas a la vez, no una sola.

Llegar en dos pasos

El 0 y el 2 no están conectados directamente: la casilla de la fila cero y la columna dos vale cero. Pero la fila cero tiene un uno en la columna uno, y la fila uno tiene un uno en la columna dos, así que existe el camino 0 → 1 → 2.

Así funciona la sugerencia de a quién seguir: gente conectada con tus conexiones, pero no contigo.

Dónde se usa

- Redes sociales: la sugerencia de a quién seguir sale de buscar conexiones a dos pasos.
- Mapas y rutas: en lugar de unos y ceros, la casilla guarda la distancia entre dos ciudades.
- Recomendaciones: filas de usuarios y columnas de productos, con la compra marcada en la casilla.
- Dependencias entre módulos de un programa, que es como se detectan las referencias circulares.

Caso de uso: imágenes y píxeles

En escala de grises cada casilla guarda la intensidad de un píxel, de 0 para negro a 255 para blanco. La fila es la coordenada vertical y la columna la horizontal, con la fila cero arriba: es el mismo orden del framebuffer.

```
 240 240 240 240  20  20        ██████
 240  20  20  20  20  20        ██
 240 240 240  20  20  20        ████
 240  20  20  20  20  20        ██
 240 240 240 240  20  20        ██████
  20  20  20  20  20  20
```

Los números de la izquierda producen la figura de la derecha: una letra E. El ciclo que la dibuja recorre la matriz casilla por casilla.

```
for (int y = 0; y < imagen.length; y++) {
    for (int x = 0; x < imagen[y].length; x++) {
        pintarPixel(x, y, imagen[y][x]);
    }
}
```

Cambiar el valor de cada casilla

Invertir los colores visita cada casilla y reemplaza su valor por el complemento: lo que valía 240 pasa a valer 15. Cada píxel se calcula solo con su propio valor, así que el resultado puede escribirse sobre la misma matriz. La figura no se mueve: los índices no cambian, solo cambia lo que guarda cada casilla.

```
for (int y = 0; y < imagen.length; y++) {
    for (int x = 0; x < imagen[y].length; x++) {
        salida[y][x] = 255 - imagen[y][x];
    }
```

```
}
```

Subir el brillo suma una constante a cada casilla y aclara la imagen completa. El resultado se recorta en 255, porque no existe un blanco más blanco; sin ese recorte un valor de 300 desbordaría el rango. Restar en lugar de sumar oscurece, y multiplicar en vez de sumar cambia el contraste.

```
for (int y = 0; y < imagen.length; y++) {
    for (int x = 0; x < imagen[y].length; x++) {
        salida[y][x] = Math.min(255, imagen[y][x] + 70);
    }
}
```

Cambiar la posición de cada casilla

El espejo horizontal no cambia ningún valor: cambia a qué casilla va cada uno. La fila se mantiene y la columna se lee al revés, empezando por la última. Es el mismo truco de índices de la diagonal inversa, aplicado a una sola dimensión.

```
int ancho = imagen[0].length;

for (int y = 0; y < imagen.length; y++) {
    for (int x = 0; x < ancho; x++) {
        salida[y][x] = imagen[y][ancho - 1 - x];
    }
}
```

Rotar noventa grados intercambia el papel de los dos índices: lo que era fila pasa a ser columna, y la nueva columna se cuenta desde el otro extremo. El resultado hay que escribirlo en una matriz nueva, porque en una imagen no cuadrada el alto y el ancho se intercambian.

```
int alto = imagen.length;

for (int y = 0; y < alto; y++) {
    for (int x = 0; x < imagen[y].length; x++) {
        salida[x][alto - 1 - y] = imagen[y][x];
    }
}
```

Mirar a los vecinos

Los filtros interesantes no dependen solo del píxel actual. Un desenfoque reemplaza cada casilla por el promedio del cuadro de tres por tres que la rodea. Los nueve valores se leen desplazando los dos índices en menos uno, cero y uno.

```
int suma = 0;

for (int dy = -1; dy <= 1; dy++) {
    for (int dx = -1; dx <= 1; dx++) {
        suma += imagen[y + dy][x + dx];
    }
}
salida[y][x] = suma / 9;
```

El resultado va en una matriz nueva: escribir sobre la original haría que los píxeles ya modificados contaminaran el cálculo de los siguientes. Cambiar la fórmula sobre esas mismas nueve casillas produce otros efectos, como detectar los bordes de la figura.

El problema del borde

Las casillas del contorno no tienen los ocho vecinos: a la fila cero le falta la fila de arriba, que no existe. Pedir la fila menos uno lanza un error de índice fuera de rango y detiene el programa.

La salida más simple es no procesar el borde, empezando el recorrido en uno y terminando antes del último índice.

```
// deja el borde sin procesar
for (int y = 1; y < alto - 1; y++) {
    for (int x = 1; x < ancho - 1; x++) {
        // aquí sí existen los ocho vecinos
    }
}
```

Otras opciones son repetir el valor del borde o tratar lo que falta como cero.

Dos operaciones, dos ideas

- Cambiar el valor: invertir y ajustar el brillo dejan cada píxel en su lugar y solo modifican lo que guarda la casilla.
- Cambiar la posición: el espejo y la rotación no tocan ningún valor, mueven los datos a otras casillas jugando con los índices.

Casi todo el procesamiento de imagen se reduce a esas dos operaciones sobre una matriz, repetidas millones de veces por segundo.

Matrices dinámicas con ArrayList anidado

El límite del arreglo nativo

El tamaño de una matriz nativa se fija al crearla, y hay problemas donde ese número no se conoce hasta que el programa ya está corriendo.

```
int[][] secciones = new int[?][?];
```

- No se sabe cuántas secciones tiene el curso ni cuántos alumnos hay en cada una hasta leer el archivo.
- Un alumno se puede inscribir a mitad de semestre, y la fila tendría que crecer.

Es el mismo problema que resolvió el `ArrayList` para una dimensión: aquí se necesita en dos.

Un ArrayList dentro de otro ArrayList

La solución es la misma idea del arreglo de arreglos: cada elemento del `ArrayList` externo es, a su vez, un `ArrayList`.

```
ArrayList<ArrayList<Integer>> matriz = new ArrayList<>();
```

El tipo se lee de afuera hacia adentro: un `ArrayList` cuyos elementos son `ArrayList` de enteros, con el tipo interno dentro de los diamantes del externo. Nace completamente vacío, sin filas y sin elementos, a diferencia del arreglo nativo que se crea con su tamaño ya reservado.

Agregar filas con add()

Cada fila es un `ArrayList` independiente que hay que crear, llenar y después agregar al externo.

```
ArrayList<ArrayList<Integer>> matriz = new ArrayList<>();

ArrayList<Integer> fila = new ArrayList<>();
fila.add(80);
fila.add(75);
fila.add(90);

matriz.add(fila);
```

El `add()` del `ArrayList` interno agrega un número a la fila, y el del externo agrega la fila completa a la matriz. Olvidar ese último `add()` es el error más común: la fila se llena pero nunca se conecta, y la matriz queda vacía.

Acceder a elementos con get()

El acceso baja un nivel a la vez, igual que los dos corchetes del arreglo nativo. El primer `get` entrega la fila y el segundo el elemento.

```
int valor = matriz[1][2];              // arreglo nativo
int valor = matriz.get(1).get(2);      // ArrayList anidado
```

Como `matriz.get(1)` devuelve un `ArrayList` completo, se le puede aplicar cualquier método de `ArrayList`. Para modificar se usa `set()` en el nivel interno: `matriz.get(1).set(2, 95)`.

Recorrer un ArrayList de ArrayList

El for-each externo entrega una fila completa y el interno cada uno de sus elementos.

```
for (ArrayList<Integer> fila : matriz) {
    for (int nota : fila) {
        System.out.print(nota + " ");
    }
    System.out.println();
}
```

El tipo del ciclo externo es `ArrayList<Integer>` y no `Integer`, porque cada elemento de la matriz es una fila entera. Con índices se usa `size()` en lugar de `length`: `matriz.size()` da las filas y `matriz.get(i).size()` las columnas de esa fila.

Filas de distinto largo

Cada sección de un curso tiene un número distinto de alumnos, y ese número cambia durante la inscripción.

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

ArrayList<String> seccionA = new ArrayList<>();
seccionA.add("Ana");
seccionA.add("Luis");

ArrayList<String> seccionB = new ArrayList<>();
seccionB.add("Carlos");
seccionB.add("Marta");
seccionB.add("Sofia");

secciones.add(seccionA);
secciones.add(seccionB);
```

Aquí `secciones.get(0).size()` es 2 y `secciones.get(1).size()` es 3. La estructura irregular sale gratis: nadie obliga a que las filas midan lo mismo. Si llega un alumno nuevo, `secciones.get(0).add("Pedro")` hace crecer esa fila sin tocar las demás.

Arreglo nativo contra ArrayList anidado

| Criterio | `int[][]` | `ArrayList` de `ArrayList` | |---|---|---| | Tamaño | Fijo al crearse | Crece y se encoge | | Filas | Existen desde el inicio | Se agregan con `add()` | | Acceso | `matriz[i][j]` | `matriz.get(i).get(j)` | | Tamaño de la fila | `matriz[i].length` | `matriz.get(i).size()` | | Contenido | Primitivos u objetos | Solo objetos |

- Arreglo nativo cuando el tamaño se conoce y no cambia: tableros, imágenes, matrices de tamaño fijo.

- `ArrayList` anidado cuando las filas se agregan durante la ejecución o la estructura cambia.

Errores comunes

- Confundir el orden de los índices. En `matriz[i][j]` el primero es la fila y el segundo la columna. Invertirlos en una matriz cuadrada no da error, solo resultados incorrectos, y por eso es difícil de detectar.
- Usar `matriz[0].length` con filas de largos distintos. Provoca un error de índice fuera de rango en cuanto una fila posterior sea más corta.
- Olvidar crear las filas después de `new int[3][]`. El arreglo externo queda lleno de referencias nulas, y usarlas lanza una excepción de puntero nulo.
- Escribir el resultado de un filtro sobre la misma matriz que se está leyendo, cuando el cálculo depende de los vecinos.
- Salirse del borde en recorridos que miran vecinos. Hay que respetar los límites o verificar cada índice antes de usarlo.
- Creer que asignar una matriz a otra variable la copia. Solo se copia la referencia, así que las dos variables apuntan a la misma estructura.

Buenas prácticas

- Nombrar los índices según su significado. Usar `fila` y `columna`, o `y` y `x` en imágenes, comunica más que `i` y `j`.
- Recorrer siempre con `matriz[i].length` en el ciclo interno. Funciona igual en matrices rectangulares y protege de errores en las irregulares.
- Guardar el alto y el ancho en variables al inicio del método cuando se usan muchas veces.

- Encapsular la matriz dentro de una clase en lugar de pasarla suelta entre métodos. La clase le da nombre a las operaciones y protege la estructura.
- Considerar una clase con atributos cuando el problema pide más de tres dimensiones.