Arrays

Programación Orientada a Objetos (CC2008) - 2026

Arrays

Semestre 02, 2026

El problema de las variables sueltas

Guardar cinco notas con cinco variables funciona, pero no escala:

```
int nota1 = 80;
int nota2 = 75;
int nota3 = 90;
int nota4 = 85;
int nota5 = 70;
```

Con 500 notas el programa se vuelve imposible de escribir. Además no se puede recorrer un conjunto de variables con nombres distintos: no hay forma de decir "dame la variable número `i`".

Un arreglo resuelve las dos cosas: un solo nombre para todo el conjunto y un número que identifica cada posición.

Definición de arreglo

Un arreglo es una estructura que almacena varios elementos del mismo tipo en posiciones contiguas de memoria, bajo un solo nombre.

```
int[] notas = {80, 75, 90, 85, 70};
```

```
                          notas
         ┌──────┬──────┬──────┬──────┬──────┐
Índice:  │  0   │  1   │  2   │  3   │  4   │
         ├──────┼──────┼──────┼──────┼──────┤
         │  80  │  75  │  90  │  85  │  70  │
         └──────┴──────┴──────┴──────┴──────┘
```

Tres propiedades definen a un arreglo en Java:

- Es homogéneo: todos sus elementos son del mismo tipo.
- Es de tamaño fijo: una vez creado no crece ni se reduce.
- Es indexado: se llega a cada elemento por su posición, empezando en cero.

Declaración y creación

Son dos operaciones distintas, aunque casi siempre se escriben juntas.

Paso 1 — declarar

```
int[] numeros;
```

Se anuncia que existe una variable que va a apuntar a un arreglo de enteros. Todavía no hay espacio reservado: la variable vale `null`.

Paso 2 — crear

```
numeros = new int[5];
```

`new` reserva memoria para cinco enteros y devuelve la referencia a ese bloque.

Paso 3 — la forma común

```
int[] numeros = new int[5];
```

Las dos operaciones en una sola línea. Es la forma que se usa en la práctica.

La alternativa: literal de arreglo

```
int[] notas = {80, 75, 90, 85, 70};
```

El tamaño se deduce de la cantidad de valores. No lleva `new` porque el compilador lo agrega por nosotros.

Esta sintaxis solo sirve en la línea de la declaración:

```
int[] numeros;
numeros = {1, 2, 3};              // error de compilación
```

Fuera de ahí hay que decir el tipo de forma explícita:

```
numeros = new int[] {1, 2, 3};    // correcto
```

Valores por defecto

Crear un arreglo con `new` no lo deja vacío: lo llena con el valor por defecto del tipo.

```
int[] numeros = new int[5];
System.out.println(numeros[0]);   // 0
```

```
┌─────┬─────┬─────┬─────┬─────┐
│  0  │  0  │  0  │  0  │  0  │
└─────┴─────┴─────┴─────┴─────┘
```

Tipo	Valor por defecto
`int`, `short`, `long`, `byte`	`0`
`double`, `float`	`0.0`
`boolean`	`false`
`char`	`'\0'`
Cualquier objeto (`String`, `Perro`, ...)	`null`

La última fila es la que más problemas causa, y es el puente hacia los arreglos de objetos.

Índices y acceso

El índice es la posición del elemento. Empieza en `0` y llega hasta `length - 1`.

```
notas[2] = 90;                      // escribir
System.out.println(notas[2]);       // leer  → 90
```

```
                 notas[2] = 90
                       │
                       ▼
```

```
┌──────┬──────┬──────┬──────┬──────┐
│  0   │  0   │  90  │  0   │  0   │
└──────┴──────┴──────┴──────┴──────┘
   0      1      2      3      4
```

Una expresión `notas[2]` se comporta exactamente como una variable de tipo `int`: se puede leer, asignar, pasar a un método o usar en una operación.

La propiedad length

Todo arreglo conoce su propio tamaño:

```
System.out.println(notas.length);   // 5
```

No es un método: no lleva paréntesis. Es un atributo público y de solo lectura.

La última posición válida siempre es `length - 1`:

```
System.out.println(notas[notas.length - 1]);   // 70
```

```
┌──────┬──────┬──────┬──────┬──────┐
│  80  │  75  │  90  │  85  │  70  │
└──────┴──────┴──────┴──────┴──────┘
   0      1      2      3      4
│◄──────────── length = 5 ──────────►│
                              último = 5 - 1 = 4
```

Pedir `notas[5]` lanza `ArrayIndexOutOfBoundsException` en tiempo de ejecución. El compilador no lo detecta porque el índice puede ser cualquier expresión.

Recorrido con el ciclo for

El ciclo tradicional usa el índice, así que sirve tanto para leer como para escribir.

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

```
   i=0    i=1    i=2    i=3    i=4
    ▼      ▼      ▼      ▼      ▼
 ┌──────┬──────┬──────┬──────┬──────┐
 │  80  │  75  │  90  │  85  │  70  │
 └──────┴──────┴──────┴──────┴──────┘
```

Escribir `i <= notas.length` es el error clásico: en la última vuelta el índice vale 5 y el programa revienta.

Recorrido con for-each

Cuando solo se necesita leer, el for-each es más limpio:

```
for (int nota : notas) {
    System.out.println(nota);
}
```

Se lee como "para cada nota dentro de notas". La variable `nota` es una copia del elemento, así que asignarle un valor no modifica el arreglo.

	for tradicional	**for-each**
Da el índice	Sí	No
Permite escribir en el arreglo	Sí	No (en primitivos)

	for tradicional	for-each
Permite recorrer al revés	Sí	No
Legibilidad	Media	Alta

Población dinámica con Scanner

El caso realista: el arreglo se llena con datos que da el usuario.

```
Scanner teclado = new Scanner(System.in);

int[] notas = new int[5];

for (int i = 0; i < notas.length; i++) {
    System.out.print("Nota " + (i + 1) + ": ");
    notas[i] = teclado.nextInt();
}
```

Se imprime `i + 1` porque las personas cuentan desde uno, aunque el arreglo cuente desde cero.

Algoritmo práctico: promedio

```
int suma = 0;

for (int nota : notas) {
    suma += nota;
}

double promedio = (double) suma / notas.length;
```

```
System.out.println(promedio);
```

Con `{80, 75, 90, 85, 70}` la suma es 400 y el promedio 80.0.

El `(double)` es obligatorio: sin él, `400 / 5` sería una división entera y perdería los decimales en cualquier caso que no sea exacto.

Arreglos de distintos tipos

Cualquier tipo puede formar un arreglo, pero cada arreglo guarda un solo tipo.

```
int[]     edades  = {18, 20, 22};
double[]  precios = {10.5, 20.7, 15.2};
boolean[] estados = {true, false, true};
String[]  nombres = {"Ana", "Luis", "Leo"};
```

La regla de oro: un arreglo almacena elementos de un solo tipo. Intentar meter un `String` en un `int[]` es un error de compilación, no un error en ejecución.

Arreglos como parámetros

Los arreglos son objetos en Java. Al pasar uno a un método no se copia su contenido: se pasa la referencia. El parámetro y la variable original son un alias del mismo arreglo.

```
public void imprimir(int[] arreglo) {
    for (int valor : arreglo) {
        System.out.println(valor);
    }
}
```

```
valores  ──────►  ┌───┬───┬───┐
                  │ 1 │ 2 │ 3 │
arreglo  ──────►  └───┴───┴───┘
```

Por eso cualquier modificación hecha dentro del método afecta al arreglo original:

```
public void modificar(int[] datos) {
    datos[0] = 999;
}

int[] valores = {1, 2, 3};

modificar(valores);

System.out.println(valores[0]);   // 999
```

Para copiar

Cuando el método no debe tocar los datos de quien lo llama, se le pasa una copia:

```
import java.util.Arrays;

int[] original = {1, 2, 3, 4};
int[] copia = Arrays.copyOf(original, original.length);
```

La copia vive en su propio espacio de memoria, así que cambiarla no afecta al original.

El salto: arreglos de objetos

Hasta aquí los arreglos guardaban valores. También pueden guardar objetos.

Antes, con variables independientes:

```
Perro perro1 = new Perro("Max", 3);
Perro perro2 = new Perro("Rocky", 5);
Perro perro3 = new Perro("Luna", 2);
```

Ahora, con un arreglo unificado:

```
Perro[] perros = new Perro[3];
```

La sintaxis es idéntica a la de los primitivos. Lo que cambia es qué guarda cada casilla: no guarda un objeto, guarda una referencia a un objeto.

Estado inicial: null

```
Perro[] perros = new Perro[3];
```

```
        ┌─────────┬─────────┬─────────┐
        │  null   │  null   │  null   │
        └─────────┴─────────┴─────────┘
```

Crear el arreglo no crea los objetos. Reserva tres casillas capaces de apuntar a un `Perro`, y las tres arrancan apuntando a nada.

Es el mismo patrón de la tabla de valores por defecto: el valor por defecto de cualquier tipo objeto es `null`.

Instanciación y el mapa de referencias

Cada objeto se crea por separado y se guarda su referencia en una casilla.

```
perros[0] = new Perro("Max", 3);
perros[1] = new Perro("Rocky", 5);
perros[2] = new Perro("Luna", 2);
```

```
     arreglo                    heap
   ┌──────────┐
 0 │    ●─────┼──────────►  Perro { nombre="Max",   edad=3 }
   ├──────────┤
 1 │    ●─────┼──────────►  Perro { nombre="Rocky", edad=5 }
   ├──────────┤
 2 │    ●─────┼──────────►  Perro { nombre="Luna",  edad=2 }
   └──────────┘
```

El arreglo vive en un lugar y los objetos en otro. Las casillas solo contienen la dirección del objeto, igual que una variable `Perro perro1`.

Con un ciclo se puede hacer lo mismo:

```
String[] nombres = {"Max", "Rocky", "Luna"};
int[] edades = {3, 5, 2};

for (int i = 0; i < perros.length; i++) {
    perros[i] = new Perro(nombres[i], edades[i]);
}
```

Usar los objetos del arreglo

La expresión `perros[0]` es una referencia a un `Perro`, así que se le puede pedir cualquier cosa que un `Perro` sepa hacer.

```
arreglo[indice].metodo()
```

Se lee en dos tiempos: primero se busca la referencia guardada en esa posición, después se ejecuta el método sobre el objeto apuntado.

```
perros[0].ladrar();                              // acción
System.out.println(perros[1].getNombre());       // consulta
perros[2].setEdad(3);                            // modificación
```

También se puede sacar la referencia a una variable, que es exactamente lo mismo:

```
Perro primero = perros[0];
primero.ladrar();
```

`primero` y `perros[0]` son dos nombres para el mismo objeto. Es el tema de aliases aplicado a arreglos.

Recorriendo un arreglo de objetos

```
// for tradicional
for (int i = 0; i < perros.length; i++) {
    System.out.println(perros[i].getNombre());
    System.out.println(perros[i].getEdad());
}

// for-each
for (Perro perro : perros) {
    perro.mostrarInformacion();
}
```

En objetos el for-each sí permite modificar: la copia es de la referencia, no del objeto, y las dos apuntan al mismo lugar.

```
for (Perro perro : perros) {
    perro.setEdad(1);      // sí modifica los objetos del arreglo
}
```

```
for (Perro perro : perros) {
    perro = new Perro("Otro", 0);   // NO modifica el arreglo
}
```

La segunda versión solo cambia hacia dónde apunta la variable temporal.

NullPointerException

El error más común con arreglos de objetos:

```
Perro[] perros = new Perro[3];
perros[0].ladrar();     // NullPointerException
```

La casilla existe, pero apunta a `null`. Pedirle un método a nada es imposible.

Dos formas de evitarlo:

```
// llenar el arreglo antes de usarlo
for (int i = 0; i < perros.length; i++) {
    perros[i] = new Perro("Sin nombre", 0);
}

// o verificar antes de usar
for (Perro perro : perros) {
    if (perro != null) {
        perro.ladrar();
    }
}
```

Conviene distinguir los dos errores:

Error	Causa
`ArrayIndexOutOfBoundsException`	La casilla no existe
`NullPointerException`	La casilla existe pero está vacía

Ejemplo completo

```
import java.util.Scanner;

public class Registro {
    public static void main(String[] args) {
        Scanner teclado = new Scanner(System.in);

        Perro[] perros = new Perro[3];

        for (int i = 0; i < perros.length; i++) {
            System.out.print("Nombre del perro " + (i + 1) + ": ");
            String nombre = teclado.nextLine();

            System.out.print("Edad de " + nombre + ": ");
            int edad = teclado.nextInt();
            teclado.nextLine();

            perros[i] = new Perro(nombre, edad);
        }

        for (Perro perro : perros) {
            perro.mostrarInformacion();
        }
    }
}
```

El primer ciclo llena el arreglo con objetos reales; el segundo lo recorre. Separar las dos fases es lo que evita el `NullPointerException`.

Parámetros variables

Java permite que un método reciba una cantidad distinta de argumentos del mismo tipo, con la sintaxis de tres puntos:

```
public double promedio(int... numeros) {
    int suma = 0;

    for (int n : numeros) {
        suma += n;
    }

    return (double) suma / numeros.length;
}

promedio(10, 20, 30);
promedio(1, 2, 3, 4, 5, 6);
```

Dentro del método, esos argumentos ya son un arreglo normal: se recorre con for-each y tiene `length`.

Las reglas:

- Solo puede haber un parámetro variable por método.
- Debe ser el último de la lista de parámetros.
- Es la opción indicada cuando no se sabe de antemano cuántos argumentos van a llegar.

Arreglos multidimensionales

Un arreglo multidimensional es en realidad un arreglo de arreglos. En álgebra lineal a esta estructura se le llama matriz.

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

matriz[0][0] = 10;

System.out.println(matriz[0][0]);
```

```
        0    1    2    3
    ┌────┬────┬────┬────┐
  0 │ 10 │ 0  │ 0  │ 0  │
    ├────┼────┼────┼────┤
  1 │ 0  │ 0  │ 0  │ 0  │
    ├────┼────┼────┼────┤
  2 │ 0  │ 0  │ 0  │ 0  │
    └────┴────┴────┴────┘
```

- El primer índice elige la fila, es decir cuál de los arreglos internos se usa.
- El segundo índice elige la columna dentro de ese arreglo.
- El número de filas se pide con `matriz.length` y el de columnas con `matriz[0].length`.
- Recorrerla completa pide dos ciclos anidados, uno por dimensión.

Buenas prácticas

- Verificar el tamaño con `length` antes de acceder a una posición calculada.
- Instanciar los objetos de un arreglo antes de usarlos, o verificar que no sean nulos.

- Recordar que pasar un arreglo a un método entrega la referencia, así que modificarlo adentro cambia el original.
- Copiar el arreglo cuando el método no deba tocar los datos de quien lo llamó.

Síntesis: primitivos contra objetos

	Arreglo de primitivos	Arreglo de objetos
Creación	`int[] num = new int[5];`	`Perro[] p = new Perro[3];`
Qué guarda cada casilla	El valor	Una referencia
Estado inicial	`0`, `0.0`, `false`	`null`
Listo para usar tras `new`	Sí	No, falta instanciar
Acceso	`num[0]`	`p[0].metodo()`
Error típico	Índice fuera de rango	Referencia nula

Lo que comparten:

- El tamaño es fijo y se consulta con `.length`.
- El primer índice es `0` y el último `length - 1`.
- Funcionan igual con `for` y con `for-each`.

Conclusiones

- Los arreglos agrupan datos del mismo tipo de manera ordenada, bajo un solo nombre y con un índice por posición.
- Son objetos y se comportan como tales: se pasan por referencia y su tamaño queda fijo al crearlos.
- Guardan tanto valores primitivos como referencias a objetos, y la diferencia entre ambos casos está en el estado inicial.
- Son la base de las estructuras más complejas que vienen después, empezando por los arreglos dinámicos.