Unidad: Estructuras de datos básicas

Arreglos, Matrices, búsqueda y ordenamiento

Introducción a la Programación

Octubre de 2025

Arreglos en Java

Objetivo de la sesión

- Comprender qué es un arreglo y para qué se usa.
- Declarar, crear e inicializar arreglos.
- Acceder y recorrer elementos con seguridad.
- Aplicar operaciones básicas: suma, promedio, mínimo/máximo y conteos.
- Anticipar casos borde y errores frecuentes.

Motivación

- Agrupar datos del **mismo tipo** bajo un solo nombre.
- Facilitar **recorridos** y **cálculos agregados** (suma, promedio, mínimos/máximos).
- Evitar duplicación de variables sueltas y errores al manejar muchas entradas.
- **Tamaño fijo** definido al crear: manejo explícito del límite de datos.

Antes vs. Después (idea)

Sin arreglos: variables sueltas

- `n1, n2, n3, ..., n50` para notas, difícil recorrer y calcular.

Con arreglos: una estructura y un índice

- `int[] notas;` permite almacenar todas las notas y procesarlas con un bucle.
- Mismo patrón para `String` (nombres), `int` (códigos), `double` (promedios parciales), etc.

Ejemplos de uso (contextos)

- **Notas de curso:** calcular promedio general, mínimo/máximo y cantidad de aprobados.
- **Resultados deportivos:** goles por partido, conteos de partidos con ≥ 2 goles.
- **Registros simples:** códigos de estudiantes o productos para reportes rápidos.

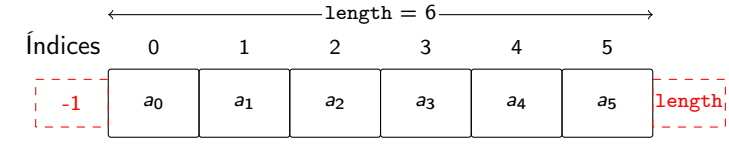
Qué es un arreglo

- Colección de elementos del **mismo tipo** accesibles por **índice**.
- Índices válidos: de `0` a `length - 1`.
- Notación de acceso: `arr[i]`.

Regla de seguridad

Usar índices siempre dentro del rango: `0 <= i < arr.length`

Arreglo: anatomía



Elemento: `arr[i]`
Índice: $0 \leq i <$ `length`
Valor: aquí como a_i (ejemplo)

Longitud y recorrido mental

- `arr.length` indica cuántos elementos contiene el arreglo.
- Idea práctica: pensar en una fila de casillas numeradas desde 0.

Ejemplo rápido

```
int[] v = {5, 8, 3, 10};
System.out.println(v.length); // 4
System.out.println(v[0]);     // 5
System.out.println(v[3]);     // 10
```

Errores frecuentes con índices

- **Fuera de rango**: acceder `arr[-1]` o `arr[arr.length]`.
- **Off-by-one**: usar `i <= arr.length - 1` junto con incrementos incorrectos.
- **Asumir longitud fija**: olvidar que `length` puede variar según el arreglo.

Recomendación

Al recorrer, preferir `for (int i = 0; i < arr.length; i++) { ... }`

Declarar y crear arreglos

- Declaración: `tipo[] nombre;`
- Creación (con tamaño): `nombre = new tipo[tamaño];`
- Ejemplo:

```
int n = 5;
int[] a;          // declaración
a = new int[n]; // creación: elementos inicializados a 0
```

- Atajo en una línea:

```
int[] b = new int[4]; // {0, 0, 0, 0}
```

Inicialización con literales

- Inicializar con una lista de valores:

```
int[] c = {10, 7, 9, 6};
String[] nombres = {"Ana", "Luis", "Mia"};
double[] medidas = {1.5, 2.0, 1.75};
char[] letras = {'A', 'B', 'C'};
boolean[] flags = {true, false, true};
```

- El tamaño queda definido por la cantidad de literales.

`length` y valores por defecto

- `arr.length` indica la cantidad de elementos.

```
int[] v = new int[3];
System.out.println(v.length); // 3
```

- Valores por defecto al crear con `new`:
  - `int`, `char`, `double`, `boolean`: inicializados (p. ej., 0, '\0', 0.0, false).
  - `String` y otros tipos de referencia: `null`.
- Recordatorio: `length` es un **atributo**, no un método (`arr.length`, no `arr.length()`).

Buenas prácticas al iniciar

- Definir el tamaño correctamente desde el inicio.
- Si usarás literales, preferir inicialización directa (`{...}`).
- Evitar accesos antes de asignar valores cuando puedan ser `null`.

Acceder a elementos por índice

- Regla: `0 <= i < arr.length`
- Lectura de un elemento y la longitud:

```
int[] v = {5, 8, 3};
System.out.println(v[0]);    // 5
System.out.println(v.length); // 3
```

- Evitar índices fuera de rango para prevenir errores.

Actualizar valores en posiciones específicas

- Asignación sobre una posición:

```
int[] v = {5, 8, 3};
v[2] = 10; // ahora v es {5, 8, 10}
```

- Uso típico: calcular con un elemento y guardar el resultado:

```
int i = 1;
v[i] = v[i] + 2; // {5, 10, 10}
```

Lectura de múltiplas posiciones (patrón común)

- Procesar varias posiciones consecutivas:

```
int[] a = {4, 7, 1, 9};
int x = a[0] + a[1]; // 4 + 7
int y = a[2] * a[3]; // 1 * 9
System.out.println(x + " " + y); // 11 9
```

- Recomendación: nombrar variables intermedias para claridad.

Buenas prácticas de acceso/actualización

- Comprobar el rango del índice antes de acceder cuando haya dudas.
- Evitar duplicar lógica: preferir funciones auxiliares si se repite el mismo patrón.
- Conservar la coherencia: si actualizas, verifica efectos en otros cálculos dependientes.

Recorrido de izquierda a derecha (`for`)

- Patrón más común para procesar todos los elementos.

```
int[] v = {5, 8, 3, 10};
for (int i = 0; i < v.length; i++) {
    System.out.println("v[" + i + "] = " + v[i]);
}
```

- Regla de seguridad: `i < v.length` (evita off-by-one).

Recorrido de derecha a izquierda (`for`)

- Útil cuando se necesita procesar primero los últimos elementos.

```
int[] v = {5, 8, 3, 10};
for (int i = v.length - 1; i >= 0; i--) {
    // procesar v[i]
}
```

- Cuidado con el límite inferior: condición `i >= 0`.

Recorrido equivalente con `while`

- Misma lógica, control de índice explícito.

```
int[] v = {5, 8, 3, 10};
int i = 0;
while (i < v.length) {
    // procesar v[i]
    i++;
}
```

- Reiniciar el índice si realizas más de un recorrido consecutivo.

Patrón con condición interna (búsqueda simple)

- Ejemplo: verificar si existe un valor objetivo.

```
int[] a = {4, 7, 1, 9};
int objetivo = 7;
boolean encontrado = false;

for (int i = 0; i < a.length; i++) {
    if (a[i] == objetivo) {
        encontrado = true;
        break; // cortar temprano
    }
}
// usar 'encontrado' después
```

- El `break` permite cortar el recorrido cuando ya no hace falta continuar.

Buenas prácticas en recorridos

- Definir con claridad el rango de índices a recorrer.
- Evitar trabajo innecesario: cortar temprano si ya se obtuvo el resultado.
- Mantener variables acumuladoras/contadores bien inicializadas antes del bucle.
- Verificar que los recorridos no dependan de modificaciones peligrosas del arreglo durante la iteración.

Suma y promedio

- Acumulación con bucle y cálculo de promedio seguro.

```
int[] notas = {60, 70, 80, 50};
int suma = 0;
for (int i = 0; i < notas.length; i++) {
    suma += notas[i];
}
double promedio = (notas.length == 0) ? 0.0 : (suma * 1.0) / notas.lengt
System.out.println("Promedio = " + promedio);
```

- Evitar división entera convirtiendo a `double`.

Mínimo y máximo con índice

- Proteger el caso de arreglo vacío.

```
int[] x = {9, 3, 7, 3, 10};
if (x.length == 0) {
    System.out.println("Arreglo vacío");
} else {
    int min = x[0], iMin = 0;
    int max = x[0], iMax = 0;
    for (int i = 1; i < x.length; i++) {
        if (x[i] < min) { min = x[i]; iMin = i; }
        if (x[i] > max) { max = x[i]; iMax = i; }
    }
    System.out.println("Min=" + min + " en i=" + iMin);
    System.out.println("Max=" + max + " en i=" + iMax);
}
```

Conteos por condición

- Ejemplo de múltiples condiciones en un mismo recorrido.

```
int[] goles = {2, 0, 1, 3, 1, 0};
int ceros = 0, mayoresIgual2 = 0;

for (int i = 0; i < goles.length; i++) {
    if (goles[i] == 0) ceros++;
    if (goles[i] >= 2) mayoresIgual2++;
}
System.out.println("Sin goles: " + ceros);
System.out.println("Con >=2 goles: " + mayoresIgual2);
```

- Útil para métricas rápidas sin estructuras adicionales.

Impresión de elementos (patrón simple)

- Recorrer e imprimir todos los elementos en orden.

```
int[] a = {4, 7, 1, 9};
for (int i = 0; i < a.length; i++) {
    System.out.println(a[i]);
}
```

- Para una sola línea con espacios:

```
for (int i = 0; i < a.length; i++) {
    if (i > 0) System.out.print(" ");
    System.out.print(a[i]);
}
System.out.println();
```

Buenas prácticas en operaciones

- Inicializar acumuladores y contadores antes del recorrido.
- Usar un solo recorrido cuando sea posible (eficiencia práctica).
- Validar el caso de arreglo vacío antes de calcular mínimo, máximo o promedio.
- Mantener consistencia en el formato de salida si se compararán resultados automáticamente.

Patrón: primero el tamaño, luego los valores

- Entrada esperada: un entero n seguido de n valores del mismo tipo.
- Ventaja: el tamaño define el arreglo y permite recorridos seguros.

```
import java.util.*;
public class Main {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int n = sc.nextInt();       // tamaño
        int[] a = new int[n];       // crear arreglo de tamaño n
        for (int i = 0; i < a.length; i++) {
            a[i] = sc.nextInt();    // leer los n valores
        }
        int suma = 0;
        for (int i = 0; i < a.length; i++) suma += a[i];
        System.out.println(suma);
        sc.close();
    }
}
```

Variantes útiles de lectura

- Lectura con condición: solo números no negativos.

```
int n = sc.nextInt();
int[] v = new int[n];
for (int i = 0; i < v.length; i++) {
    int x = sc.nextInt();
    if (x < 0) x = 0;   // ajuste simple
    v[i] = x;
}
```

- Confirmación básica posterior a la carga:

```
for (int i = 0; i < v.length; i++) {
    if (i > 0) System.out.print(" ");
    System.out.print(v[i]);
}
System.out.println();
```

Buenas prácticas al leer datos

- Definir primero n y crear el arreglo con ese tamaño.
- Recorrer exactamente `0..n-1` para asignar todos los elementos.
- Si hay reglas de validación, aplicarlas antes de guardar en el arreglo.
- Mantener un formato de salida consistente cuando se requiera comparar resultados automáticamente.

Casos borde al trabajar con arreglos

- **Arreglo vacío** (`length == 0`): proteger antes de mínimo, máximo y promedio.
- **Un solo elemento**: validarlo explícitamente (evita ciclos innecesarios).
- **Datos duplicados**: decidir si se requiere el primer índice, el último o todos.
- **Valores atípicos**: confirmar reglas (p. ej., ignorar negativos, truncar a rango válido).

Errores típicos y cómo evitarlos

- **Off-by-one**: usar `i < arr.length`, no `i <= arr.length`.

```
for (int i = 0; i < a.length; i++) {
    // OK
}
```

- **Índices fuera de rango**: nunca acceder a `a[-1]` ni `a[a.length]`.

```
int i = a.length - 1;  // último índice válido
```

- **División entera en promedios**: convertir al menos un operando a `double`.

```
double prom = (a.length == 0) ? 0.0 : (suma * 1.0) / a.length;
```

Checklist rápido antes de correr

- ¿Todos los recorridos usan `i < arr.length`?
- ¿Se verifica `length == 0` cuando corresponde?
- ¿Acumuladores/contadores están inicializados?
- ¿Promedios calculados como `double`?
- ¿El formato de salida es consistente con lo esperado?

Ejemplos (Notas de curso)

Entrada: n y luego n notas (0–100). **Salida:** min & índice, max & índice, promedio, aprobados (≥ 51).

```
int n = sc.nextInt();
int[] notas = new int[n];
for (int i = 0; i < n; i++) notas[i] = sc.nextInt();

if (n == 0) System.out.println("Sin datos");
else {
  int min = notas[0], iMin = 0, max = notas[0], iMax = 0, suma = 0, aprob = 0;
  for (int i = 0; i < n; i++) {
    int x = notas[i];
    if (x < min) { min = x; iMin = i; }
    if (x > max) { max = x; iMax = i; }
    suma += x; if (x >= 51) aprob++;
  }
  double prom = (suma * 1.0) / n;
  System.out.println(min + " " + iMin);
  System.out.println(max + " " + iMax);
  System.out.println(prom);
  System.out.println(aprob);
}
```

Ejemplos (Goles por partido)

Entrada: m y luego m enteros (goles). **Salida:** imprimir valores, contar $= 0$ y ≥ 2.

```
int m = sc.nextInt();
int[] g = new int[m];
for (int i = 0; i < m; i++) g[i] = sc.nextInt();

int sin = 0, con2 = 0;
for (int i = 0; i < m; i++) {
  if (i > 0) System.out.print(" ");
  System.out.print(g[i]);
  if (g[i] == 0) sin++;
  if (g[i] >= 2) con2++;
}
System.out.println();
System.out.println(sin);
System.out.println(con2);
```

Resumen de la sesión

- Arreglos: declaración, creación, `length` y acceso por índice.
- Recorridos con `for`/`while` en ambos sentidos.
- Operaciones básicas: suma, promedio, mínimo/máximo con índice, conteos por condición.
- Lectura de datos con patrón “tamaño + valores”.
- Casos borde y errores frecuentes: off-by-one, índices fuera de rango, división entera.

Recursos de la sesión

- Código fuente: `https://gist.github.com/helderfernandez/9d63cb5cd792fece58c7e3a91c7bb493`
- Archivos incluidos:
  - `S1_01_DeclaracionCreacionInicializacion.java`
  - `S1_02_AccesoActualizacion.java`
  - `S1_03_Recorridos.java`
  - `S1_04_OperacionesBasicas.java`
  - `S1_05_LecturaTamanoValores.java`
  - `S1_06A_Ejemplo_NotasCurso.java`, `S1_06B_Ejemplo_GolesPartido.java`