



# UNIDAD 4. ARRAYS


Programación  
CFGS DAW

## Nomenclatura

A lo largo de este tema se utilizarán distintos símbolos para distinguir elementos importantes dentro del contenido. Estos símbolos son:

 Importante

 Atención

 Interesante

## ÍNDICE DE CONTENIDO

<b>1. Introducción.....</b>	<b>4</b>
<b>2. Propiedades.....</b>	<b>4</b>
<b>3. Vectores (Arrays unidimensionales).....</b>	<b>5</b>
3.1 Declaración.....	5
3.2 Instancia.....	5
3.3 Almacenamiento.....	6
3.4 Longitud de un vector.....	6
3.5 Recorrido de un vector.....	7
3.6 Copia de vectores.....	8
<b>4. Arrays multidimensionales.....</b>	<b>9</b>
<b>5. La clase Arrays.....</b>	<b>10</b>
<b>6. La clase String.....</b>	<b>11</b>
6.1 Comparación.....	11
6.2 Métodos más utilizados.....	12
6.3 Lectura con Scanner.....	16
<b>7. Búsqueda en Vectores.....</b>	<b>17</b>
7.1 Búsqueda secuencial.....	17
7.2 Búsqueda dicotómica o binaria.....	18
<b>8. Ordenación de vectores.....</b>	<b>19</b>
<b>9. Ejemplo de llenado y recorrido de un vector.....</b>	<b>20</b>
<b>10. Agradecimientos.....</b>	<b>20</b>

## UD04. ARRAYS

### 1. INTRODUCCIÓN



Un **array** o **vector** es una colección de valores de un **mismo tipo** dentro de una misma variable. De forma que se puede acceder a cada valor independientemente.

Para Java, además, un array es un objeto que tiene propiedades que se pueden manipular.

Los arrays solucionan problemas concernientes al manejo de muchas variables que se refieren a datos similares.

Por ejemplo si tuviéramos la necesidad de almacenar las notas de una clase con 18 alumnos, necesitaríamos 18 variables, con la tremenda lentitud de manejo que supone eso. Solamente calcular la nota media requeriría una tremenda línea de código. Almacenar las notas supondría al menos 18 líneas de código.

En lugar de crear 18 variables sería mucho mejor crear un array de tamaño 18 (es como si tuviéramos una sola variable que puede almacenar varios valores).

Gracias a los arrays se puede crear un conjunto de variables con el mismo nombre. La diferencia será que un número (índice del array) distinguirá a cada variable.

### 2. PROPIEDADES

Algunas propiedades de los arrays son:

- Los arrays se utilizan como contenedores para almacenar datos relacionados (en lugar de declarar variables por separado para cada uno de los elementos del array).
- **Todos los datos incluidos en el array son del mismo tipo.** Se pueden crear arrays de enteros de tipo *int* o de reales de tipo *float*, pero **en un mismo array no se pueden mezclar tipos de datos**, por ej. *int* y *float*.
- El tamaño del array se establece cuando se crea el array (con el operador *new*, igual que cualquier otro objeto).
- A los elementos del array se accederá a través de la posición que ocupan dentro del conjunto de elementos del array.
- Los arrays unidimensionales se conocen con el nombre de **vectores**.
- Los arrays bidimensionales se conocen con el nombre de **matrices**.

### 3. VECTORES (ARRAYS UNIDIMENSIONALES)

#### 3.1 Declaración

Un array se declara de forma similar a una variable simple pero añadiendo corchetes para indicar que se trata de un array y no de una variable simple del tipo especificado.

Un Vector (array unidimensional) se puede declarar de dos formas:

- *tipo identificador[];*
- *tipo[] identificador;*

Donde **tipo** es el tipo de dato de los elementos del vector e **identificador** es el nombre de la variable.

Ejemplos:

```
int notas[];  
double cuentas[];
```

Declara un array de tipo int y otro de tipo double. Esta declaración indica para qué servirá el array, pero no reserva espacio en la memoria RAM al no saberse todavía el tamaño del mismo. Todavía no puede utilizarse el array, falta instanciarlo.

#### 3.2 Instancia

**Tras la declaración del array, se tiene que instanciar**, para ello se utiliza el operador **new**, que es el que realmente crea el array indicando un tamaño. Cuando se usa **new**, es cuando se reserva el espacio necesario en memoria. Un array no inicializado es un array **null** (sin valor).

Ejemplo:

```
int notas[]; // Declaramos 'notas' como array de tipo int  
notas = new int[3]; // Instanciamos 'notas' a tamaño 3  
  
// Es habitual declarar e instanciar en una sola línea  
int notas[] = new int[3];
```

En el ejemplo anterior se crea un array de tres enteros (con los tipos básicos se crea en memoria el array y se inicializan los valores, los números se inicializan a 0).

### 3.3 Almacenamiento

Los valores del array se asignan (almacenan) utilizando el índice del mismo entre corchetes.

⚡ El primer elemento del vector siempre estará en la posición o índice 0

Índices →	0	1	2	3	4
Valores →	8	10	2	3	5

Por ejemplo, para almacenar el valor 2 en la tercera posición del array escribiríamos:

```
notas[2] = 2;
```

También se pueden asignar valores al array en la propia declaración e instancia:

```
int notas[] = {8, 10, 2, 3, 5};
```

```
int notas2[] = new int[] {8, 10, 2, 3, 5}; //Equivalente a la anterior
```

Esto declara e inicializa un array de cinco elementos. El ejemplo sería equivalente a:

```
notas[0] = 8;
```

```
notas[1] = 10;
```

```
notas[2] = 2;
```

```
notas[3] = 3;
```

```
notas[4] = 5;
```

En Java (como en otros lenguajes) el primer elemento de un array está en la posición **cero**. El primer elemento del array notas, es notas[0].

📖 Se pueden declarar arrays a cualquier tipo de datos (enteros, booleanos, doubles, ... e incluso objetos).

### 3.4 Longitud de un vector

Los arrays poseen una propiedad llamada **length** que indica su tamaño.

Ejemplo:

```
int notas[] = new int[4]; // Declara e instancia vector tipo int de tamaño 4
```

```
System.out.println( notas.length ); // Mostrará un 4
```

Si el vector tiene como en el ejemplo 4 elementos, la propiedad *length* nos devolverá el valor entero 4, pero su primer elemento se encuentra en notas[0] y el último en notas[3].

### 3.5 Recorrido de un vector

Para recorrer un vector (acceder a todos sus elementos) siempre será necesario un bucle.

En el siguiente ejemplo declaramos e instanciamos un vector tipo `int` con las notas de un alumno y luego utilizamos un bucle `for` para recorrer el vector y mostrar todos los elementos.

```
// Declaramos e instanciamos vector tipo int
int notas[] = new int[] {7, 3, 9, 6, 5};

// Como el vector es de tamaño 5 sus elementos estarán en las posiciones de 0 a 4
// Recorremos el vector desde i=0 hasta i<5 (es decir, desde 0 hasta 4)
for (int i = 0; i < notas.length; i++) {
    System.out.println(notas[i]);
}
```

Ahora vamos a calcular la nota media (sumar todas y luego dividir entre el número de notas):

```
// Declaramos suma y media
int suma = 0;
int media;

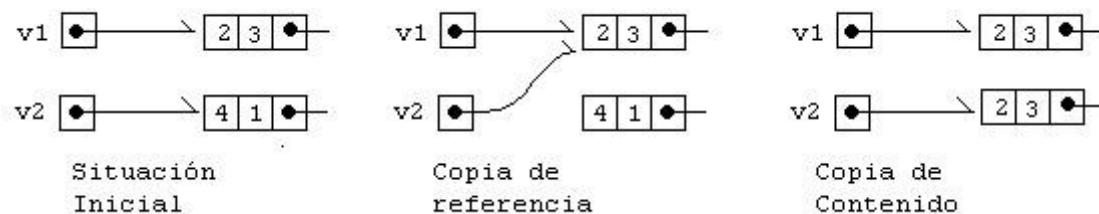
// Recorremos el vector desde 0 hasta 4, acumulando las notas en suma
for (int i = 0; i < notas.length; i++) {
    suma += notas[i]; // Equivale a: suma = suma + notas[i]
}

// Calculamos la media y la mostramos por pantalla
media = suma / notas.length;
System.out.println("La nota media es: " + media);
```

### 3.6 Copia de vectores

Para copiar vectores no basta con igualar un vector a otro como si fuera una variable simple.

Si partimos de dos vectores v1, v2 e hiciéramos v2=v1, lo que ocurriría sería que v2 apuntaría a la posición de memoria de v1. Eso es lo que se denomina un copia de referencia:



Si por ejemplo queremos copiar todos los elementos del vector v2 en el vector v1, existen dos formas para hacerlo:

- **Copiar los elementos uno a uno**

```
for (i = 0; i < v1.length; i++)  
    v2[i] = v1[i];
```

- **Utilizar la función arraycopy**

**System.arraycopy(v\_origen, i\_origen, v\_destino, i\_destino, length);**

*v\_origen: Vector origen*

*i\_origen: Posición inicial de la copia*

*v\_destino: Vector destino*

*i\_destino: Posición final de la copia*

*length: Cantidad de elementos a copiar*

*// Copiamos todos los elementos de v1 en v2*

*System.arraycopy(v1, 0, v2, 0, v1.length);*

## 4. ARRAYS MULTIDIMENSIONALES

Los arrays pueden tener más de una dimensión. Los más utilizados son los **arrays de 2 dimensiones**, conocidos como **matrices**.



Las matrices se definen de las siguientes formas:

```
tipo identificador[][];
```

```
tipo[][] identificador;
```

Por ejemplo, declaramos e instanciamos un array de 3 x 3 (3 filas x 3 columnas).

```
double precios[][] = new int[3][3];
```

Accedemos a sus valores utilizando dobles corchetes.

```
precios[0][0] = 7.5;
```

```
precios[0][1] = 12;
```

```
precios[0][2] = 0.99;
```

```
precios[1][0] = 4.75;
```

```
// etc.
```

		Columnas		
		0	1	2
Filas	0	(0,0)	(0,1)	(0,2)
	1	(1,0)	(1,1)	(1,2)
	2	(2,0)	(2,1)	(2,2)

Veamos otro ejemplo en el que declaramos e instanciamos un array de 3 filas x 6 columnas para almacenar las notas de 3 alumnos (la fila corresponde a un alumno, y cada columna a las notas de dicho alumno):

```
int notas[][] = new int[3][6]; // Es equivalente a 3 vectores de tamaño 6
```

Suponiendo que las notas ya están almacenadas, vamos a recorrer la matriz (array bidimensional) para mostrar las notas por pantalla. Hay que tener en cuenta que como tiene dos dimensiones necesitaremos un bucle anidado (uno para las filas y otro para las columnas de cada fila):

```
// Para cada fila (alumno)
```

```
for (int i = 0; i < notas.length; i++) {
```

```
    System.out.print("Notas del alumno " + i + ": ");
```

```
    // Para cada columna (nota)
```

```
    for (int j = 0; j < notas[i].length; j++) {
```

```
        System.out.print(notas[i][j] + " ");
```

```
    }
```

```
}
```



## 5. LA CLASE ARRAYS

En el paquete *java.util* se encuentra una clase estática llamada **Arrays**. Esta clase estática permite ser utilizada como si fuera un objeto (como ocurre con *Math*). Esta clase posee **métodos** muy interesantes para utilizar sobre arrays.

Su uso es:

```
Arrays.método(argumentos);
```

Algunos métodos son:

- **fill**: permite rellenar todo un array unidimensional con un determinado valor. Sus argumentos son el array a rellenar y el valor deseado:

Por ejemplo, llenar un array de 23 elementos enteros con el valor -1

```
int valores[] = new int[23];
```

```
Arrays.fill(valores,-1); // Almacena -1 en todo el array 'valores'
```

También permite decidir desde qué índice hasta qué índice rellenamos:

```
Arrays.fill(valores,5,8,-1); // Almacena -1 desde el 5º a la 7º elemento
```

- **equals** : Compara dos arrays y devuelve true si son iguales (false en caso contrario). Se consideran iguales si son del mismo tipo, tamaño y contienen los mismos valores.

```
Arrays.equals(valoresA, valoresB); // devuelve true si los arrays son iguales
```

- **sort** : Permite ordenar un array en orden ascendente. Se pueden ordenar sólo una serie de elementos desde un determinado punto hasta un determinado punto.

```
int x[]={4,5,2,3,7,8,2,3,9,5};
```

```
Arrays.sort(x); // Ordena x de menor a mayor
```

```
Arrays.sort(x,2,5); // Ordena x solo desde 2º al 4º elemento
```

- **binarySearch** : Permite buscar un elemento de forma ultrarrápida en un array ordenado. Devuelve el índice en el que está colocado el elemento buscado. Ejemplo:

```
int x[]={1,2,3,4,5,6,7,8,9,10,11,12};
```

```
Arrays.sort(x);
```

```
System.out.println(Arrays.binarySearch(x,8)); //Devolvería 7
```

## 6. LA CLASE STRING

Las cadenas de texto son objetos especiales. Los textos deben manejarse creando objetos de tipo *String*.

Ejemplo:

```
String texto1 = "¡Prueba de texto!";
```

Las cadenas pueden ocupar varias líneas utilizando el operador de concatenación "+".

```
String texto2 = "Este es un texto que ocupa " +  
                "varias líneas, no obstante se puede "+  
                "perfectamente encadenar";
```

También se pueden crear objetos *String* sin utilizar constantes entrecomilladas, usando otros constructores:

```
char[] palabra = {'P', 'a', 'l', 'a', 'b', 'r', 'a'}; // Array de char  
String cadena = new String(palabra);
```

### 6.1 Comparación

⚡ Los objetos *String* NO pueden compararse directamente con los operadores de comparación == como las variables simples

En su lugar se deben utilizar estos métodos:

- ***cadena1.equals(cadena2)***. El resultado es true si la cadena1 es igual a la cadena2. Ambas cadenas son variables de tipo *String*.
- ***cadena1.equalsIgnoreCase(cadena2)***. Como la anterior, pero en este caso no se tienen en cuenta mayúsculas y minúsculas.
- ***s1.compareTo(s2)***. Compara ambas cadenas, considerando el orden alfabético. Si la primera cadena es mayor en orden alfabético que la segunda, devuelve la diferencia positiva entre una cadena y otra, si son iguales devuelve 0 y si es la segunda la mayor, devuelve la diferencia negativa entre una cadena y otra. Hay que tener en cuenta que el orden no es el del alfabeto español, sino que usa la tabla ASCII, en esa tabla la letra ñ es mucho mayor que la o.
- ***s1.compareToIgnoreCase(s2)***. Igual que la anterior, sólo que además ignora las mayúsculas.

## 6.2 Métodos más utilizados

Son funciones que poseen los propios objetos de tipo *String*. Para utilizarlos basta con poner el nombre del método y sus parámetros después del nombre del objeto *String*.

```
objetoString.método(argumentos);
```

Algunos de los métodos más utilizados son:

**valueOf** : Convierte valores que no son de cadena a forma de cadena.

```
String numero = String.valueOf(1234); // Convierte el número int 1234 en el String "1234"
```

**length** : Devuelve la longitud de una cadena (el número de caracteres de la cadena):

```
String texto1="Prueba";  
System.out.println(texto1.length()); // Escribe un 6
```

**Concatenar cadenas** : Se puede hacer de dos formas, utilizando el método *concat* o con el operador *+*.

```
String s1= "Buenos ", s2= " días", s3, s4;  
s3 = s1 + s2;  
s4 = s1.concat(s2);
```

En ambos casos el contenido de *s3* y *s4* sería el mismo: "Buenos días".

**charAt** : Devuelve un carácter concreto de la cadena. El carácter a devolver se indica por su posición (el primer carácter es la posición 0). Si la posición es negativa o sobrepasa el tamaño de la cadena, ocurre un error de ejecución, una excepción tipo *IndexOutOfBoundsException-Exception* (recuerda este tipo de error, se repetirá muchas veces).

```
String s1="Prueba";  
char c1 = s1.charAt(2); // c1 valdrá 'u'
```

**substring** : Da como resultado una porción del texto de la cadena. La porción se toma desde una posición inicial hasta una posición final (sin incluir esa posición final). Si las posiciones indicadas no son válidas ocurre una excepción de tipo *IndexOutOfBoundsException-Exception*. Se empieza a contar desde la posición cero.

```
String s1="Buenos días";  
String s2=s1.substring(7,10); // s2 = "día"
```

**indexOf** : Devuelve la primera posición en la que aparece un determinado texto en la cadena. En el caso de que la cadena buscada no se encuentre, devuelve -1. El texto a buscar puede ser *char* o *String*.

```
String s1="Quería decirte que quiero que te vayas";  
System.out.println(s1.indexOf("que")); // Devuelve 15
```

También se puede buscar desde una determinada posición:

```
String s1="Quería decirte que quiero que te vayas";  
System.out.println(s1.indexOf("que",16)); // Ahora devolvería 26
```

**lastIndexOf** : Devuelve la última posición en la que aparece un determinado texto en la cadena. Es casi idéntica a la anterior, sólo que busca desde el final.

```
String s1="Quería decirte que quiero que te vayas";  
System.out.println(s1.lastIndexOf("que")); //Devolvería 26
```

También permite comenzar a buscar desde una determinada posición.

**endsWith** : Devuelve true si la cadena termina con un determinado texto.

```
String s1="Quería decirte que quiero que te vayas";  
System.out.println(s1.endsWith("vayas")); //Devolvería true
```

**startsWith** : Devuelve true si la cadena empieza con un determinado texto.

```
String s1="Quería decirte que quiero que te vayas";  
System.out.println(s1.startsWith("vayas")); // Devolvería false
```

**replace**: Cambia todas las apariciones de un carácter (o caracteres) por otro/s en el texto que se indique y lo almacena como resultado. El texto original no se cambia, por lo que hay que asignar el resultado de replace a un *String* para almacenar el texto cambiado.

#### Ejemplo1

```
String s1="Mariposa";  
System.out.println(s1.replace('a', 'e')); //Devuelve "Meripose"  
System.out.println(s1); //Sigue valiendo "Mariposa"
```

Para guardar el valor deberíamos hacer:

```
String s2 = s1.replace('a','e');
```

#### Ejemplo2

```
String s1="Buscar armadillos";  
System.out.println(s1.replace("ar","er")); //Devuelve "Buscer ermadillos"  
System.out.println(s1); //Sigue valiendo "Buscar armadillos"
```

**toUpperCase** : Obtiene la versión en mayúsculas de la cadena. Es capaz de transformar todos los caracteres nacionales:

```
String s1 = "Batallón de cigüeñas";  
System.out.println(s1.toUpperCase()); //Escribe: BATALLÓN DE CIGÜEÑAS
```

**toLowerCase** : Obtiene la versión en minúsculas de la cadena.

**ToCharArray** : Consigue un array de caracteres a partir de una cadena. De esa forma podemos utilizar las características de los arrays para manipular el texto, lo cual puede ser interesante para manipulaciones complicadas.

```
String s="texto de prueba";  
char c[]=s.toCharArray();
```

**format** : Modifica el formato de la cadena a mostrar. Muy útil para mostrar sólo los decimales que necesitemos de un número decimal. Indicaremos "%" para indicar la parte entera más el número de decimales a mostrar seguido de una "f" :

```
System.out.println(String.format("%.2f", number)); // Muestra el número con dos decimales.
```

**matches** : Examina la expresión regular que recibe como parámetro (en forma de *String*) y devuelve verdadero si el texto que examina cumple la expresión regular. Una expresión regular es una expresión textual que utiliza símbolos especiales para hacer búsquedas avanzadas.

Las expresiones regulares pueden contener:

- Caracteres. Como a, s, ñ,... y les interpreta tal cual. Si una expresión regular contuviera sólo un carácter, matches devolvería verdadero si el texto contiene sólo ese carácter. Si se ponen varios, obliga a que el texto tenga exactamente esos caracteres.
- Caracteres de control (\n,\\,...)
- Opciones de caracteres. Se ponen entre corchetes. Por ejemplo [abc] significa a, b ó c.
- Negación de caracteres. Funciona al revés impide que aparezcan los caracteres indicados. Se pone con corchetes dentro de los cuales se pone el carácter circunflejo (^). [^abc] significa ni a ni b ni c.
- Rangos. Se ponen con guiones. Por ejemplo [a-z] significa: cualquier carácter de la a a la z.
- Intersección. Usa &&. Por ejemplo [a-x&&r-z] significa de la r a la x (intersección de ambas expresiones).
- Sustracción. Ejemplo [a-x&&[^cde]] significa de la a a la x excepto la c, d ó e.
- Cualquier carácter. Se hace con el símbolo punto (.)
- Opcional. El símbolo ? sirve para indicar que la expresión que le antecede puede aparecer una o ninguna veces. Por ejemplo a? indica que puede aparecer la letra a o no.
- Repetición. Se usa con el asterisco (\*). Indica que la expresión puede repetirse varias veces o incluso no aparecer.
- Repetición obligada. Lo hace el signo +. La expresión se repite una o más veces (pero al menos una).
- Repetición un número exacto de veces. Un número entre llaves indica las veces que se repite la expresión. Por ejemplo \d{7} significa que el texto tiene que llevar siete números (siete cifras del 0 al 9). Con una coma significa al menos, es decir \d{7,} significa al menos siete veces (podría repetirse más veces). Si aparece un segundo número indica un máximo número de veces \d{7,10} significa de siete a diez veces.

Veamos algunos ejemplos:

```

28 public static void pruebas_matches(String[] args) {
29
30     String cadena="Solo se que no se nada";
31
32     // ejemplo1: devolvera false, ya que la cadena tiene mas caracteres
33     System.out.println("ejemplo1: "+cadena.matches("Solo"));
34
35     // ejemplo2: devolvera true, siempre y cuando no cambiemos la cadena Solo
36     System.out.println("ejemplo2: "+cadena.matches("Solo.*"));
37
38     // ejemplo3: devolvera true, siempre que uno de los caracteres se cumpla
39     System.out.println("ejemplo3: "+cadena.matches(".*[qnd].*"));
40
41     // ejemplo3: devolvera false, ya que ninguno de esos caracteres estan
42     System.out.println("ejemplo4: "+cadena.matches(".*[xyz].*"));
43
44     // ejemplo4: devolvera true, ya que le indicamos que no incluya esos caracteres
45     System.out.println("ejemplo4: "+cadena.matches(".*[^xyz].*"));
46
47     // ejemplo5: devolvera true, si quitamos los caracteres delante de ? del String original seguira devolviendo true
48     System.out.println("ejemplo5: "+cadena.matches("So?lo se qu?e no se na?da"));
49
50     // ejemplo6: devolvera false, ya que tenemos una S mayuscula empieza en el String
51     System.out.println("ejemplo6: "+cadena.matches("[a-z].*"));
52
53     // ejemplo7: devolvera true, ya que tenemos una S mayuscula empieza en el String
54     System.out.println("ejemplo7: "+cadena.matches("[A-Z].*"));
55
56     String cadena2="abc1234";
57
58     // ejemplo8: devolvera true, ya que minimo debe repetirse alguno de los caracteres al menos una vez
59     System.out.println("ejemplo8: "+cadena2.matches("[abc]+.*"));
60
61     // ejemplo9: devolvera true, ya que, ademas del ejemplo anterior, indicamos que debe repetirse un valor numerico 4 veces
62     System.out.println("ejemplo9: "+cadena2.matches("[abc]+\\d{4}"));
63
64     // ejemplo10: devolvera true, ya que, ademas del ejemplo anterior, indicamos que debe repetirse un valor numerico entre 1 y 10 veces
65     System.out.println("ejemplo10: "+cadena2.matches("[abc]+\\d{1,10}"));
66

```

### 6.3 Lectura con Scanner

Como ya sabemos, la lectura de un *String* utilizando la clase *Scanner* se realiza con el método *nextLine()*:

```

Scanner in = new Scanner(System.in);
String s = in.nextLine();

```

Si leemos un tipo de dato numérico, entero por ejemplo, antes de leer un *String* deberemos limpiar el buffer de entrada, de lo contrario leerá el valor '\n' (salto de línea) introducido después del número y se lo asignará a la variable *String*, con lo que no se leerá bien la entrada.

Deberemos hacer lo siguiente:

```

Scanner in = new Scanner(System.in);
System.out.print("Introduce un número: ");
int n = in.nextInt();
in.nextLine(); // Limpiamos el buffer de entrada
System.out.print("Introduce un String: ");
String s = in.nextLine();

```

## 7. BÚSQUEDA EN VECTORES

Existen dos formas de buscar un elemento dentro de un vector: la búsqueda secuencial y la búsqueda dicotómica o binaria.

### 7.1 Búsqueda secuencial

La búsqueda secuencial es la más fácil de las dos ya que consiste en comparar los elementos del vector con el elemento a buscar.

Un ejemplo es el siguiente, donde se devuelve la posición del elemento en el vector y si no lo encuentra, devuelve el valor -1:

```
56      public static int busquedaSecuencial(int[] v, int elemento)
57      {
58          int i, posicion = -1;
59
60          for(i = 0; i < v.length && posicion == -1; i++)
61              if(v[i] == elemento)
62                  posicion = i;
63
64          return posicion;
65      }
```



## 7.2 Búsqueda dicotómica o binaria

En este caso el vector debe estar ordenado. Se dividirá en dos para buscar el elemento en una parte del vector o en otra y así sucesivamente hasta encontrar, o no, el elemento.

Un ejemplo es el siguiente:

```
66 public static int busquedaDicotomica(int[] v, int elemento)
67 {
68     int izq = 0; // El índice 'izq' se establece en la posición 0
69     int der = v.length-1; // El índice 'der' se establece en la última posición
70     int centro = (izq + der)/2; // El índice 'centro' se establece en la posición central
71     int posicion;
72
73     while(izq <= der && v[centro] != elemento)
74     {
75         if(elemento < v[centro])
76             der = centro - 1; // Si el elemento es menor que el centro cambiamos el índice 'der'
77         else
78             izq = centro + 1; // Sino cambiamos el índice 'izq'
79
80         centro = (izq + der)/2; // Actualizamos el centro
81     }
82
83     if(izq > der)
84         posicion = -1;
85     else
86         posicion = centro;
87
88     return posicion;
89 }
```

En este caso, al igual que en el anterior, la función devuelve la posición del elemento o -1 si no lo encuentra. Es importante ver que el vector debe estar ordenado para quedarnos con la parte desde la izquierda al centro del vector o desde el centro del vector a la derecha, dependiendo si el elemento a buscar es mayor o menor al elemento del centro del vector.

Esta búsqueda es más óptima que la secuencial ya que no tiene que recorrer el vector entero.

## 8. ORDENACIÓN DE VECTORES

Existen diferentes algoritmos para ordenar un vector. Algunos ejemplos son:

- Burbuja
- Inserción
- Selección
- Quicksort (el más rápido de los cuatro)

No es necesario saber cómo funcionan estos algoritmos ya que Java ya los tiene implementados y podemos utilizar métodos de ordenación muy fácilmente (por ejemplo con `Arrays.sort`).

De todos modos es un tema fascinante. Quien tenga interés puede encontrar la explicación y el código en diferentes lenguajes en la siguiente página web:

[https://en.wikibooks.org/wiki/Algorithm\\_Implementation/Sorting](https://en.wikibooks.org/wiki/Algorithm_Implementation/Sorting)

## 9. EJEMPLO DE LLENADO Y RECORRIDO DE UN VECTOR

Vamos a ver un ejemplo de cómo llenar y mostrar un vector de enteros:

```
19 public static void main(String[] args) {  
20     Scanner in = new Scanner(System.in);  
21  
22     int[] vector = new int[5]; // Creación de un vector de enteros de tamaño 5  
23     int i;  
24  
25     System.out.print("Introduce los valores del vector: ");  
26  
27     // Llenado del vector con valores desde teclado  
28     for(i = 0; i < vector.length; i++)  
29         vector[i] = in.nextInt();  
30  
31     System.out.print("El vector introducido es: ");  
32  
33     // Mostrar el vector  
34     for(i = 0; i < vector.length; i++)  
35         System.out.print(vector[i] + " ");  
36  
37     System.out.println();  
38 }  
39 }
```

Hacemos uso de la propiedad **length**. También podríamos haber puesto un 5 (tamaño del vector).

La salida es:

```
run:  
Introduce los valores del vector: 1 2 3 4 5  
El vector introducido es: 1 2 3 4 5  
BUILD SUCCESSFUL (total time: 4 seconds)
```

Podemos introducir los valores con espacios y una vez introducido el quinto número darle al 'intro'.  
O introducir un valor por línea.