



# Tablas

## Objetivos

- Conocer las tablas, que permiten almacenar múltiples valores en una variable.
- Crear tablas de distinto tipo y longitudes.
- Utilizar las operaciones básicas que se emplean con las tablas.
- Diseñar programas que hagan uso de tablas, donde se almacenan los datos necesarios.
- Modificar la longitud de una tabla en tiempo de ejecución sin pérdida de los datos que contiene.
- Usar la API de Java relacionada con las tablas y aplicar su uso a la resolución de problemas.

## Contenidos

- 5.1. Variables escalares versus tablas
- 5.2. Índices
- 5.3. Construcción de tablas
- 5.4. Referencias
- 5.5. Uso de tablas
- 5.6. Tablas como parámetros de funciones
- 5.7. Operaciones con tablas: la clase Arrays
- 5.8. Tablas  $n$ -dimensionales

# Introducción

Responde a una sencilla pregunta: ¿cuántos valores puede almacenar simultáneamente una variable? Según vimos en la primera unidad, la respuesta es obvia: un solo valor en cada instante. Analicemos el siguiente código:

```
edad = 6;
edad = 23;
```

La variable `edad` inicialmente almacena el valor 6 y a continuación 23. Cada nueva asignación modifica `edad`, pero, independientemente del número de asignaciones, la variable contiene tan solo un valor en cada momento. A este tipo de variables (que solo pueden almacenar un valor de forma simultánea) se les conoce como *variables escalares*.

¿Existe una forma de almacenar más de un valor simultáneamente en una variable? La respuesta es sí, mediante el uso de tablas.

## Argot técnico



En la bibliografía es común encontrar autores que denominan a las tablas *arrays* (su nombre en inglés) o *vectores*.

También es usual encontrar para las tablas el nombre de *arreglos*, que es una mala traducción al castellano de *array*. No se recomienda su uso.

## 5.1. Variables escalares versus tablas

Una tabla es una variable que permite guardar más de un valor simultáneamente. Podemos ver una tabla como una «supervariable» que engloba a otras variables, llamadas *elementos* o *componentes* referidas con un mismo nombre, con la condición de que todas sean del mismo tipo. En la Figura 5.1 se representa la tabla `edad` que guarda los valores —años— de los asistentes a una fiesta: 85, 3, 19, 23 y 7.

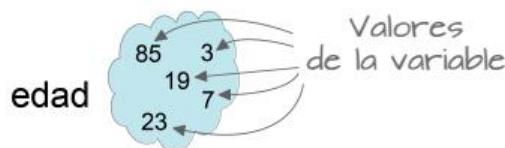
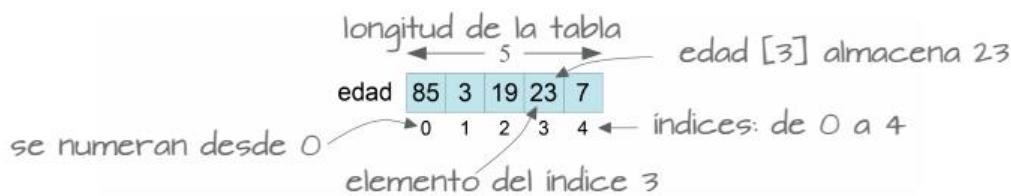


Figura 5.1. Representación de una tabla con varios valores.

Siempre que necesitemos manejar varios datos del mismo tipo simultáneamente, en general, se recomienda utilizar una tabla en lugar de varias variables escalares. Es mucho más cómodo trabajar con una tabla que almacena, por ejemplo, 100 valores, que hacerlo con 100 variables escalares. Cuando desconocemos cuántos datos son necesarios gestionar, no existe otra alternativa que utilizar tablas, ya que, a la hora de crearlas, el número de elementos que guardaremos en ella se puede elegir dinámicamente.

## ■ 5.2. Índices

El problema es cómo distinguir entre cada uno de los elementos o componentes que constituyen una tabla. En nuestro caso, ¿cómo podemos utilizar el valor 85 o el valor 19 que se encuentran almacenados en la tabla edad? La solución consiste en asignar un número de orden a cada elemento, para así poder diferenciarlos. A este número se le llama *índice*. Al primer elemento se le asigna el índice 0, al segundo el índice 1 y así sucesivamente. Al último elemento le corresponde como índice el número total de elementos menos uno.



**Figura 5.2.** Una tabla de cinco elementos enteros. El hecho de comenzar a numerar los elementos en 0 provoca que el último elemento sea 4 (la longitud de la tabla menos uno).

La forma de utilizar un elemento concreto de una tabla es por medio del nombre de la variable que identifica a la tabla junto al número —índice— entre corchetes (`[ ]`) que distingue ese elemento. Por ejemplo, para utilizar el cuarto elemento de la tabla `edad` —elemento con índice 3—, escribiremos `edad[3]`, que contiene un valor de 23.

Veamos un ejemplo de cómo mostrar y asignar un elemento:

```
System.out.println(edad[0]); //muestra el primer elemento: 85
edad[3] = 8; //asigna un nuevo valor al cuarto elemento
```

### ■ 5.2.1. Índices fuera de rango

La variable `edad` es una tabla de 5 enteros y podemos utilizar cada uno de los cinco elementos que la componen de la forma: `edad[0]`, `edad[1]`..., `edad[4]`.

¿Qué ocurrirá si utilizamos un índice que se encuentra fuera del rango de 0 a 4? Es decir, ¿qué efectos producen `edad[-2]` o `edad[7]`? En ambos casos obtendremos un error en tiempo de ejecución que provoca que el programa termine de forma inesperada, ya que se detecta que los elementos con los índices utilizados no existen. La mayoría de los errores al trabajar con tablas provienen de utilizar índices fuera de rango. Se recomienda prestar especial atención a esto.

## ■ 5.3. Construcción de tablas

En el momento de crear una tabla, deberemos tener en cuenta lo siguiente:

- Decidir qué tipo de datos vamos a almacenar y cuántos elementos necesitamos.
- Declarar una variable para la tabla.
- Crear la propia tabla.

### ■ ■ ■ 5.3.1. Longitud y tipo

Una tabla se define mediante dos características fundamentales: su longitud y su tipo. La longitud es el número de elementos que tiene, y el tipo de una tabla es el de los datos que almacena en todos y cada uno de sus elementos. En la Figura 5.3 podemos ver dos tablas: la primera compuesta por tres elementos de tipo `double` y la segunda por seis elementos de tipo `int`.

<table border="1" style="border-collapse: collapse; width: 100px;"> <tr><td style="padding: 2px;">1.27</td></tr> <tr><td style="padding: 2px;">0</td></tr> <tr><td style="padding: 2px;">1</td></tr> <tr><td style="padding: 2px;">2</td></tr> </table>	1.27	0	1	2	<table border="1" style="border-collapse: collapse; width: 100px;"> <tr><td style="padding: 2px;">0.3</td></tr> <tr><td style="padding: 2px;">1</td></tr> <tr><td style="padding: 2px;">2</td></tr> </table>	0.3	1	2	<table border="1" style="border-collapse: collapse; width: 100px;"> <tr><td style="padding: 2px;">9.0</td></tr> <tr><td style="padding: 2px;">2</td></tr> </table>	9.0	2	longitud 3 tipo: double								
1.27																				
0																				
1																				
2																				
0.3																				
1																				
2																				
9.0																				
2																				
<table border="1" style="border-collapse: collapse; width: 100px;"> <tr><td style="padding: 2px;">5</td></tr> <tr><td style="padding: 2px;">1</td></tr> <tr><td style="padding: 2px;">2</td></tr> <tr><td style="padding: 2px;">3</td></tr> <tr><td style="padding: 2px;">4</td></tr> <tr><td style="padding: 2px;">5</td></tr> </table>	5	1	2	3	4	5	<table border="1" style="border-collapse: collapse; width: 100px;"> <tr><td style="padding: 2px;">-1</td></tr> <tr><td style="padding: 2px;">0</td></tr> <tr><td style="padding: 2px;">2</td></tr> <tr><td style="padding: 2px;">0</td></tr> <tr><td style="padding: 2px;">-7</td></tr> </table>	-1	0	2	0	-7	<table border="1" style="border-collapse: collapse; width: 100px;"> <tr><td style="padding: 2px;">0</td></tr> <tr><td style="padding: 2px;">1</td></tr> <tr><td style="padding: 2px;">2</td></tr> <tr><td style="padding: 2px;">3</td></tr> <tr><td style="padding: 2px;">4</td></tr> <tr><td style="padding: 2px;">5</td></tr> </table>	0	1	2	3	4	5	longitud 6 tipo: int
5																				
1																				
2																				
3																				
4																				
5																				
-1																				
0																				
2																				
0																				
-7																				
0																				
1																				
2																				
3																				
4																				
5																				

Figura 5.3. Tablas con distintas longitudes y tipos.

En la Figura 5.3, la primera tabla podría utilizarse para almacenar, por ejemplo, el tiempo empleado en realizar algunas pruebas, mientras que la segunda tabla podría usarse para guardar el número de puntos obtenidos en distintas partidas de un videojuego.

### ■ ■ ■ 5.3.2. Variables de tabla

El primer paso para crear una tabla es declarar la variable utilizando corchetes (`[]`), símbolo que diferencia entre una variable escalar y una que es tabla. Es posible utilizar dos sintaxis equivalentes:

```
tipo nombreVariable[];  
tipo[] nombreVariable;
```

donde `tipo` representa cualquier tipo primitivo. Veamos cómo declarar la variable `edad` como una tabla de tipo `int`:

```
int edad[];
```

o mediante la forma (ambas son equivalentes):

```
int [] edad;
```

En este punto la variable está declarada, pero no hemos construido ninguna tabla.

#### Argot técnico



En esta unidad solo crearemos tablas de algún tipo primitivo. Sin embargo, en Java podemos crear tablas de más tipos, por ejemplo: podríamos crear una tabla de tipo `Scanner`. De la forma:

```
Scanner tablaScanner[] = new Scanner [] ();
```

En la unidad sobre clases, veremos cómo usar y crearlas con distintos tipos de objetos.

Y en la Unidad 12 estudiaremos un mecanismo alternativo a las tablas para almacenar colecciones de datos.

### ■ ■ ■ 5.3.3. Operador new

Una vez que hemos declarado una variable, crearemos una tabla con la longitud adecuada y la asignaremos a la variable. La sintaxis es:

```
nombreVariable = new tipo[longitud];
```

Veamos cómo crear la tabla `edad`, de tipo `int` con una longitud de 5 elementos:

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

El operador `new` construye una tabla donde todos los elementos se inicializan a 0, para tipos numéricos, o `false` si la tabla es booleana.

#### Argot técnico



Los elementos de las tablas de otros tipos se inicializan a `null`. Véase el Apartado 5.4.2.

Es posible declarar la variable y crear la tabla en una única sentencia.

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

Existe una alternativa para crear una tabla sin necesidad de utilizar el operador `new`. En la misma declaración se asignan valores a los elementos de la tabla, que se crea con la longitud necesaria para albergar todos los valores asignados. Por ejemplo:

```
int datos[] = {2, -3, 0, 7}; //tabla de longitud 4
```

Esta sentencia declara la variable `datos` y crea una tabla de cuatro enteros, donde cada elemento tiene asignado el valor correspondiente, equivalente a:

```
int datos[]; //declaramos la variable  
datos = new int[4]; //creamos la tabla  
datos[0] = 2; //asignamos valores  
datos[1] = -3;  
datos[2] = 0;  
datos[3] = 7;
```

La creación de una tabla mediante la asignación de valores solo puede realizarse en la misma instrucción donde se declara. No se puede escribir

```
int datos[];  
datos = {2, -3, 0, 7} //Error! Solo en la declaración
```

## ■ 5.4. Referencias

La siguiente instrucción:

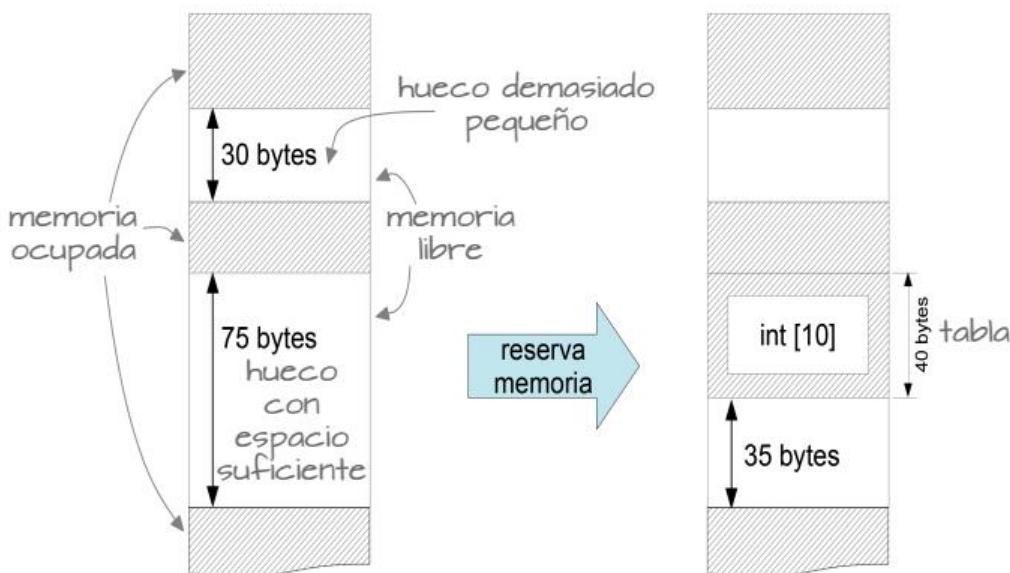
```
edad = new int[10];
```

construye una tabla de 10 elementos de tipo `int` y la asigna a la variable `edad`. Estudie-  
mos cómo funciona el operador `new`:

1. En primer lugar, calcula el tamaño físico de la tabla, es decir, el número de bytes que ocupará la tabla en la memoria. Este cálculo es sencillo y se obtiene de multiplicar la longitud de la tabla por el tamaño de su tipo. Como ejemplo vamos a calcular el tamaño físico de una tabla de longitud diez de tipo `int`:

$$\text{tamaño en memoria} = n.^{\circ} \text{ elementos tabla} \times \text{tamaño tipo (int)} = 10 \times 4 \text{ bytes} = 40 \text{ bytes}$$

2. Conociendo el tamaño físico de la tabla, busca en la memoria un hueco libre (memoria no utilizada) con un tamaño suficiente para albergar todos los elementos de la tabla consecutivamente (véase la Figura 5.4).
3. Reserva la memoria necesaria para almacenar la tabla y la marca como memoria ocupada, siendo este el sitio donde se almacenarán los elementos de la tabla.
4. Por último, recorre todos los elementos de la tabla inicializándolos de la siguiente manera: 0 si es una tabla numérica, `false` si la tabla es booleana.



**Figura 5.4.** Reserva de memoria. Representación de la memoria antes y después de construir una tabla de 10 enteros.

En este punto hemos creado la tabla. El siguiente paso es asignarla a la variable correspondiente; para ello Java dispone de un mecanismo para indicar dónde está la tabla en la memoria. Cada posición de la memoria de un ordenador tiene una dirección única que la identifica. Así, la primera posición de memoria tiene la dirección 000; la siguiente, la dirección 001, y así sucesivamente. En Java a cada dirección de memoria se le denomina *referencia*.

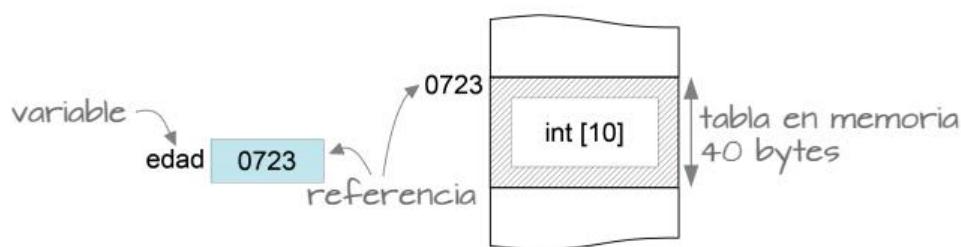
### Argot técnico



Realmente una referencia es algo un poco más sofisticado, pero con esta simplificación es suficiente para entender el concepto.

La forma de que una variable sepa dónde está la tabla en la memoria es asignándole la referencia de la primera posición que ocupa (una tabla puede ocupar varias posiciones consecutivas). Lo que almacenan realmente las variables de tabla son referencias. Por

ese motivo se las conoce también como *variables de referencia*. La Figura 5.5 muestra la zona de la memoria reservada para la nueva tabla, que empieza en la dirección, por ejemplo, 0723. Por tanto, esa es la referencia de la tabla.



**Figura 5.5.** Asignación de una referencia a una variable. La variable `edad` contiene la referencia que le permite acceder a una posición de memoria y encontrar ahí la tabla con todos sus datos.

Si ejecutamos las siguientes líneas:

```
int t[] = new int[10];
System.out.println(t);
```

podríamos pensar (erróneamente) que se muestra por consola el valor de cada elemento de la tabla `t`, pero esto no es así. Lo que se muestra es la referencia que guarda la variable `t`, que suele tener una forma similar a: `I@659e0bfd`.

### Argot técnico



La «`I`» de la referencia especifica que la tabla es de enteros (`int`). El primer carácter identifica el tipo de la tabla: `B` para `byte`, `D` para `double`, `Z` para booleanos, etcétera.

A partir de `@` se muestra la dirección de memoria (en hexadecimal) en la que se encuentra la tabla. En nuestro ejemplo: `659e0bfd`.

Las referencias se modifican en cada ejecución, dependiendo de la ocupación de la memoria. En las representaciones gráficas es mucho más intuitivo sustituir los valores de la referencia por una flecha, que tiene el mismo significado: «la variable está referenciando esta tabla». Esta representación se muestra en la Figura 5.6, donde también hemos cambiado el bloque de memoria por casillas que representan los elementos de la tabla.



**Figura 5.6.** Representación de una tabla referenciada por una variable. La representación más habitual es mediante una flecha, ya que de forma gráfica se aprecia perfectamente a qué tabla referencia cada variable.

### Actividad propuesta 5.1

Crea tres tablas de cinco elementos: la primera de enteros, la segunda de `double` y la tercera de booleanos. Muestra las referencias en las que se encuentra cada una de las tablas anteriores.

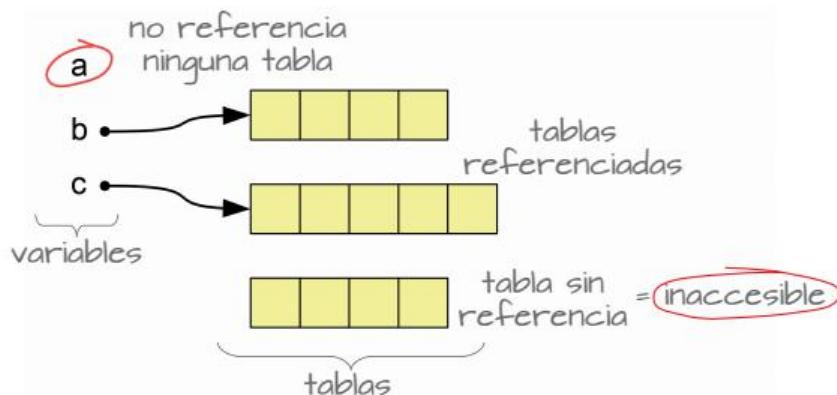
Ahora que entendemos cómo funcionan las referencias, podemos analizar el siguiente código:

```
int a[], b[], c[]; //variables
b = new int[4]; //tabla de cuatro enteros accesible mediante la variable b
c = new int[5]; //tabla de cinco enteros accesible mediante la variable c
new int[3]; //Creamos una tabla cuya referencia no se asigna a ninguna variable
```

A pesar de que la variable `a` está declarada, por sí sola no sirve de nada, ya que no referencia ninguna tabla, es decir, no disponemos de ningún elemento para almacenar datos. Por el contrario, las variables `b` y `c` sí son útiles, ya que referencian sendas tablas.

La última instrucción (`new int[3];`) construye una tabla con tres elementos enteros, pero la referencia que devuelve `new` no se asigna a ninguna variable, lo que convierte la tabla en inútil, ya que es inaccesible. Una vez que hemos perdido la referencia de una tabla, no existe forma alguna de recuperarla.

La Figura 5.7 muestra todos los casos posibles de referencias.

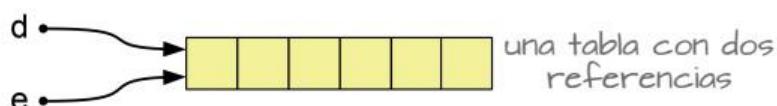


**Figura 5.7.** La tabla que no está referenciada tan solo ocupa espacio en la memoria y es completamente imposible acceder a sus datos. Java se encarga de liberar la memoria ocupada de forma inútil.

Las variables pueden verse como medios para acceder a las tablas a las que referencian. Es posible acceder a una misma tabla mediante más de una variable; para ello, la tabla debe estar referenciada por estas variables.

La Figura 5.8 representa el resultado del siguiente código:

```
int d[], e[]; //variables
d = new int[6]; //construimos una tabla referenciada por d
e = d; //ahora la variable e referencia la misma tabla que d. Ambas guardan
//la misma dirección de memoria
```



**Figura 5.8.** Tabla multirreferenciada. En programación es muy habitual utilizar más de una variable que refieran los mismos elementos. Este mecanismo es la base para compartir información entre distintas partes de un programa.

## Actividad propuesta 5.2

Construye una tabla de 10 elementos del tipo que deseas. Declara diferentes variables de tabla que referenciarán la tabla creada. Comprueba, imprimiendo por pantalla, que todas las variables contienen la misma referencia.

### Argot técnico



Hay autores que utilizan una analogía con las referencias: una televisión con varios mandos a distancia. Piensan en las variables como mandos a distancias que permiten manejar y utilizar los datos de una tabla (la televisión).

Ahora podemos acceder a los mismos datos utilizando la variable `d` o la variable `e`: utilizar `d[2]` es equivalente a utilizar `e[2]`, ya que `d` y `e` referencian la misma tabla. De todas formas esta práctica es poco aconsejable, ya que puede producir cambios no deseados en la tabla. La única condición para que una variable pueda referenciar a una tabla es que el tipo de ambas coincidan. El siguiente código es erróneo debido a que los tipos no coinciden:

```
boolean t1[]; //variable para tablas booleanas
int t2[]; //variable para tablas enteras
t1 = new boolean[10]; //construye y asigna una tabla de 10 booleanos
t2 = t1; //;ERROR! Tipos incompatibles
//t2 puede referenciar tablas enteras, pero no booleanas
```

### 5.4.1. Recolector de basura

¿Qué ocurre cuando una tabla no está referenciada por ninguna variable? Lo primero y más obvio es que dicha tabla es inútil; no hay forma de acceder a sus elementos. Pero existe un segundo problema: la tabla está ocupando espacio en la memoria. Quizá el tamaño de unas pocas tablas inútiles en la memoria no sea significativo, pero una aplicación puede dejar, durante su ejecución, grandes cantidades de memoria ocupada inaccesible. Esto puede ocurrir por un mal diseño o de forma malintencionada.

Java soluciona el problema mediante un mecanismo muy ingenioso: periódicamente se inicia un proceso llamado *recolector de basura* que comprueba todas las tablas construidas. Si encuentra alguna inaccesible —sin variables que la refieran— la destruye, dejando libre el espacio que estaba ocupando en la memoria.

### Argot técnico



En la bibliografía es habitual encontrar al recolector de basura denominado por su nombre en inglés: *garbage collector*.

El recolector de basura se ejecuta de forma automática, aunque nunca sabremos cuándo comenzará. Es un mecanismo que no es exclusivo de Java y que utilizan otros lenguajes.

El secreto de su funcionamiento es que Java lleva una doble contabilidad:

- Un listado de todas las tablas creadas.
- Para cada tabla, un listado de todas las variables que hacen referencia a ella.

Con esta información, va comprobando para todas las tablas si existe alguna que no tenga referencia. En este caso, destruye dicha tabla.

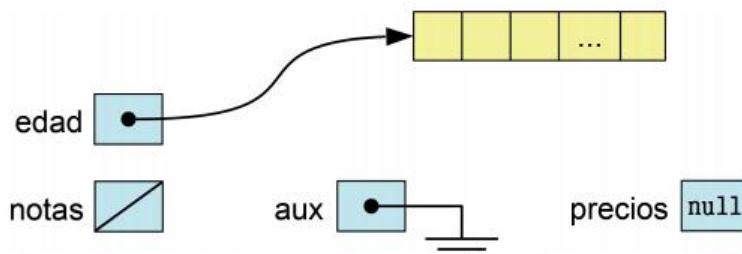
El recolector de basura no solo trabaja con tablas, como veremos en la unidad sobre clases, también se encarga de comprobar todos los objetos construidos.

## 5.4.2. Referencia null

Hemos visto la forma de asignar a una variable la referencia de una tabla: al crearla con el operador `new` o a través de otra variable. Pero existe la forma de hacer justo lo contrario, es decir, hacer que una variable que referencia a una tabla no referencie nada. Para ello disponemos del literal `null`, que significa «vacío».

Veamos un ejemplo de cómo dejar sin referencia a una tabla:

```
int t1[], t2[]; //variables de tipo tabla entera
t1 = new int[100]; //t1 referencia una tabla de 100 elementos
t2 = t1; //ahora t2 también referencia la misma tabla
t1 = null; //anulamos t1: no referencia nada
           //la tabla sigue siendo accesible desde t2
t2 = null; //anulamos t2: tampoco hace referencia a nada
//la tabla es inaccesible: el recolector de basura se encargará de ella
```



**Figura 5.9.** Mientras una referencia se suele representar con una flecha entre la variable y la tabla, es habitual encontrar la referencia vacía representada mediante una línea cruzada, una toma de tierra o incluso con la propia palabra `null`. Todo ello informa que la variable no está referenciando nada.

## 5.5. Uso de tablas

Existen distintas técnicas para trabajar con tablas: podemos considerar que solo algunos elementos de una tabla almacenan información útil mientras la información de otros elementos no es relevante (véase el Apartado 5.5.2). Otra alternativa es suponer que todos los elementos de una tabla contienen siempre información útil y es la tabla la que adapta su longitud a las necesidades del momento. Esta segunda técnica simplifica la resolución de los problemas y permite aprovechar las herramientas que proporciona Java, lo que minimiza las implementaciones propias que tenemos que desarrollar. Por estos motivos, hemos decidido que para las soluciones de las actividades se preferirá esta técnica.

Veamos un ejemplo donde todos los elementos de una tabla contienen siempre datos útiles, sea la variable `estatura` que referencia una tabla con la altura de algunas personas:

estatura	1,23	2,07	1,74	1,86	1,35
	0	1	2	3	4

Las tablas, una vez creadas, mantienen su longitud constante y no es posible cambiar el número de elementos que contienen. Si necesitamos modificar la longitud de una tabla, lo que haremos será crear una segunda tabla con el número de elementos necesarios y copiar en ella los datos que nos interesan de la tabla original. Si la nueva tabla se referencia con la misma variable que referenciaba a la original, a efectos prácticos es como si la tabla original hubiera modificado su longitud. Por lo tanto, si deseamos modificar la longitud de `estatura`, tendremos que crear una segunda tabla que estará referenciada por la misma variable `estatura`, dando la sensación de que la longitud de la tabla ha cambiado.

Indistintamente de la opción elegida, podemos optar por mantener cierto orden entre los datos de una tabla.

## ■■■ 5.5.1. Tablas ordenadas

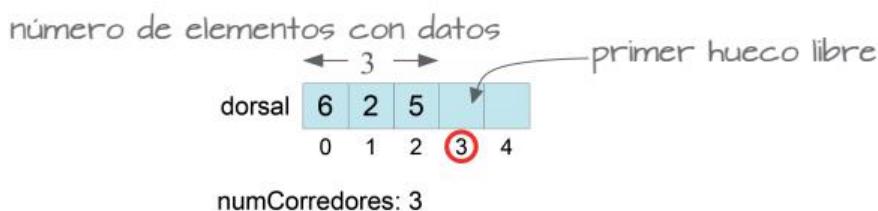
En ocasiones, interesa que los datos estén ordenados siguiendo algún criterio. Un ejemplo de una tabla ordenada en sentido creciente es:

precios	12,3	18,0	34,65	80,19	102,0
	0	1	2	3	4

Cuando usamos tablas ordenadas, es muy importante, en el momento de insertar o eliminar un elemento, realizar la operación teniendo cuidado de que la tabla continúe ordenada.

## ■■■ 5.5.2. Tablas + indicador

Otra técnica para usar las tablas es suponer que no todos sus elementos almacenan datos. La tabla estará subdividida en dos partes: la primera con los elementos que almacenan datos y la segunda con los elementos vacíos. Para ello, la tabla irá acompañada de una variable entera que funciona como indicador del número de datos que contiene la tabla (véase la Figura 5.10), es decir, el indicador especifica cuántos elementos forman la primera parte (datos útiles) y el resto se consideran elementos vacíos.



**Figura 5.10.** Tabla de longitud 5 donde solo se almacenan 3 dorsales (solo tres datos). La variable `numCorredores` actúa como indicador de la tabla. En caso de insertar un nuevo corredor se usaría el primer hueco (elemento) libre (con índice 3).

Con esta técnica, cada vez que se inserta o elimina un dato en la tabla no es necesario modificar su longitud; en su lugar se modifica el indicador, lo que hace que el número de elementos con datos aumente o disminuya.

Los elementos de una tabla siempre almacenan algún valor, por lo que los elementos que se consideran vacíos realmente contienen valores (denominados *valores basura*) que nunca tendremos en cuenta.

Aunque para esta unidad hemos optado por usar tablas en las que todos sus elementos contienen datos útiles, hemos comentado la técnica de **tabla + indicador** porque es común encontrarla en la bibliografía. Su principal ventaja es que no necesita redimensionar continuamente la tabla, pero en cambio, debemos llevar un control manual del indicador.

## 5.6. Tablas como parámetros de funciones

Recordemos cuál es el mecanismo de paso de parámetros al invocar una función: el valor de la variable que se utiliza en la llamada se copia al parámetro de la función (que es una variable local en la función). Veamos un ejemplo:

```
muestra(a); //llamada a la función
...
void muestra(int b) { //definición de la función
    ...
}
```

El valor de la variable `a` se copia en el parámetro `b`. Si en el cuerpo de la función se modifica la variable `b`, se está modificando una copia, no la variable `a`.

Cuando utilizamos tablas como parámetros el mecanismo es el mismo, es decir, el valor de la variable utilizada en la llamada se copia al parámetro de la función. Pero en este caso, lo que se copia es la referencia de una tabla, con lo que se consigue que la tabla esté referenciada tanto por la variable utilizada en la llamada como por el parámetro. La modificación de un elemento de la tabla dentro de la función es visible desde la referenciada externa a la función. Esto es normal, ya que disponemos de dos referencias, pero de una única tabla donde se realizan las modificaciones.

En el ejemplo de la Figura 5.11, si dentro del cuerpo de la función `ejemploFuncion()` ejecutamos

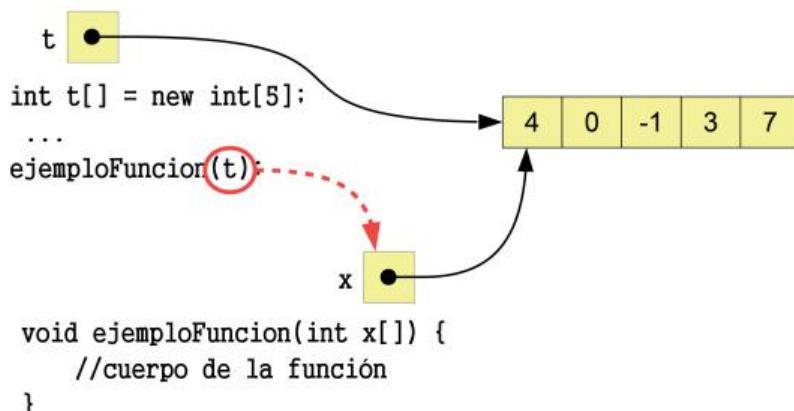
```
x[2] = 10;
```

estamos cambiando también `t[2]`, ya que tanto `x` como `t` referencian la misma tabla.

### Recuerda

Las funciones solo pueden devolver un único valor mediante la instrucción `return`. El uso de tablas como parámetros y el hecho que se comparten sus datos entre la función y quien la invoca, permite que una función pueda devolver más información que con un simple `return`.





**Figura 5.11.** La variable `t`, que referencia a la tabla de datos, copia su referencia a la variable `x` (parámetro de la función). El mecanismo de paso de parámetros es siempre el mismo: el valor de la variable (sea escalar o de tipo tabla) se copia en el parámetro. De esta forma, tanto la variable `t` como `x` referencian a la misma tabla

## ■ 5.7. Operaciones con tablas: la clase Arrays

La API de Java proporciona herramientas que facilitan el trabajo del programador; hemos usado ya las clases `Scanner` y `Math`, entre otras. También disponemos de la clase `Arrays`, que tiene una serie de métodos estáticos para operar con tablas. Se ubica en el paquete `java.util` y tiene que ser importada para poder usarse.

```
import java.util.Arrays;
```

No existe ningún motivo por el que no podamos implementar nuestras propias operaciones para tablas, pero `Arrays` nos da la seguridad de un código eficiente, sin errores y la comodidad que supone ahorrarnos el tiempo de escribir nuestra implementación. Siempre que sea posible aprovecharemos las funcionalidades presentes en la clase `Arrays`.

Las opciones al trabajar con tablas son prácticamente infinitas, pero casi todo lo que necesitamos puede sintetizarse en las operaciones que veremos en este apartado.

### ■ ■ 5.7.1. Obtención del número de elementos de una tabla

Java proporciona un mecanismo para conocer el número de elementos —longitud— con el que se construyó una tabla.

```
nombreVariable.length
```

Por ejemplo, el código:

```
int notas[] = new int [10];
System.out.println("Longitud de la tabla notas: " + notas.length);
```

muestra por pantalla:

Longitud de la tabla notas: 10

Averiguar la longitud de una tabla puede ser necesario cuando manipulamos, en el cuerpo de una función, una tabla pasada como parámetro. En este contexto desconocemos con qué longitud se creó.

## 5.7.2. Inicialización

Si deseamos inicializar con un valor distinto a los valores por defecto, tendremos que asignar a cada elemento de la tabla el valor en cuestión.

### Recuerda



Por defecto, una tabla se inicializa con valores específicos:

- 0 para tipos numéricos.
- `false` para booleanos.

Aunque todavía no lo hemos visto (se estudiará en la Unidad 7), cuando la tabla contiene objetos las tablas se inicializan a `null`.

Existe un método de la clase `Arrays` que hace exactamente esto.

- `static void fill(tipo t, tipo valor)`: que inicializa todos los elementos de la tabla `t` con `valor`. Esta función está sobrecargada, siendo posible utilizarla con cualquier tipo primitivo, representado por `tipo`, con la restricción de que el tipo de la tabla coincida con el tipo del valor pasado como parámetro.

Veamos cómo inicializar la tabla `sueldos` con un valor de 1234,56:

```
Arrays.fill(sueldos, 1234.56); //inicializa todos los elementos
```

Si interesa inicializar solo algunos elementos de una tabla, disponemos de:

- `static void fill(tipo t[], int desde, int hasta, tipo valor)`: asigna los elementos de la tabla `t`, comprendidos entre los índices `desde` y `hasta`, sin incluir este último, con `valor`. `tipo` representa cualquier tipo primitivo, con la restricción de que el tipo de la tabla coincida con el tipo del valor pasado como parámetro.

Como ejemplo, veamos cómo inicializar los elementos con índices del 3 al 6 (en la llamada utilizaremos 7, ya que no se incluye en el rango) de la tabla `sueldos`:

```
Arrays.fill(sueldos, 3, 7, 1234.56); //inicializa solo el rango 3..6
```

El rango de índices es el comprendido entre el 3 y el anterior al 7, es decir, del 3 al 6.

## 5.7.3. Recorrido

Muchas operaciones con tablas implican recorrerlas, que consiste en visitar sus elementos para procesarlos. Por *procesar* se entiende cualquier operación que realicemos con un elemento, como, por ejemplo, asignarle un valor, mostrarlo por consola o hacer algún tipo de cálculo con él. El recorrido de una tabla puede ser total, cuando se recorren todos sus

elementos o parcial, cuando solo visitamos un subconjunto de ellos. El patrón de código para recorrer una tabla es:

```
for (int i = desde; i <= hasta; i++) {  
    //procesado de t[i]  
    ...  
}
```

Se visitan los elementos con índices comprendidos entre `desde` y `hasta`. En el código anterior, el último elemento visitado es `t[hasta]`. Si deseamos que el último elemento visitado sea justo el anterior a `hasta`, bastará con modificar en el `for` la condición: `i<=hasta` por `i<hasta`.

Por ejemplo, si deseamos incrementar un 10 % todos los elementos de la tabla `sueldos`,

```
for (int i = 0; i < sueldos.length; i++) { //recorremos la tabla  
    sueldos[i] = sueldos[i] + 0.1 * sueldos[i]; //procesamos  
}
```

La instrucción `for` tiene una sintaxis alternativa, conocida como `for-each` o `for` extendido, que permite recorrer los elementos de una tabla.

```
for (declaración variable: tabla) {  
    ...  
}
```

Es necesario declarar una variable que tiene que ser del mismo tipo que la tabla. Esta variable irá tomando en cada iteración cada uno de los valores de los elementos de la tabla y el bucle se ejecutará tantas veces como elementos existan. Es importante tener en cuenta que la variable es una copia de cada elemento, y que en el caso de que se modifique, estamos modificando una copia, no el elemento de la tabla. Veamos cómo sumar todos los elementos de la tabla `sueldos`:

```
double sumaSueldos = 0;  
for (double sueldo : sueldos) { //sueldo tomará todos los valores de la tabla  
    sumaSueldos += sueldo;  
}
```

## Actividad resuelta 5.1

Crear una tabla de longitud 10 que se inicializará con números aleatorios comprendidos entre 1 y 100. Mostrar la suma de todos los números aleatorios que se guardan en la tabla.

### Solución

```
int valores[];  
valores = new int[10];  
  
//Vamos a recorrer la tabla para inicializar con valores aleatorios.  
//Cuando se modifican los elementos de una tabla no podemos usar for-each  
for (int i = 0; i < valores.length; i++) {  
    valores[i] = (int) (Math.random()*100 + 1);  
}
```

```
//Ahora recorreremos la tabla para calcular la suma de sus elementos
int suma = 0;
for(int valor: valores) {
    suma += valor;
}

System.out.println("La suma de los valores aleatorios es " + suma);
```

### Actividad propuesta 5.3

Introduce por teclado un número  $n$ ; a continuación, solicita al usuario que teclee  $n$  números. Realiza la media de los números positivos, la media de los negativos y cuenta el número de ceros introducidos.

#### 5.7.4. Mostrar una tabla

Mostrar una tabla consiste en mostrar sus elementos. Si ejecutamos

```
int t[] = {8, 41, 37, 22, 19};
System.out.println(t); //muestra una referencia
```

no se muestra el contenido de la tabla; en su lugar se muestra la referencia que contiene `t`.

Para mostrar una tabla tendremos que realizar un recorrido en el que mostrar, uno a uno, cada elemento que la compone. Esta funcionalidad la realiza el método estático `toString()` de la clase `Arrays`, que se usa en combinación con `System.out.println()`. Veamos un ejemplo:

```
int t[] = {8, 41, 37, 22, 19};
System.out.println(Arrays.toString(t));
```

Muestra los valores de la tabla entre corchetes: [8, 41, 37, 22, 19].

Si preferimos escribir nuestra propia implementación, tendremos que recorrer la tabla y mostrar sus elementos, de la siguiente forma:

```
for (i = 0; i < t.length; i++) { //recorremos toda la tabla
    System.out.println(t[i]); //mostramos cada elemento
}
```

o utilizando `for-each`:

```
for (int elemento: t) {
    System.out.println(elemento);
}
```

Evidentemente, es más cómodo utilizar `Arrays.toString()`.

## Actividad resuelta 5.2

Diseñar un programa que solicite al usuario que introduzca por teclado 5 números decimales. A continuación, mostrar los números en el mismo orden que se han introducido.

### Solución

```
/*
 * Para guardar 5 número es posible utilizar cinco variables escalares, pero es
 * mucho más cómodo una tabla con 5 elementos. Los números pueden tener
 * decimales, por lo tanto, declararemos la tabla de tipo double.
 * Tendremos que recorrer la tabla para insertar los valores.
 */
Scanner sc = new Scanner(System.in);
sc.useLocale(Locale.US); //para separar los decimales con punto (no con coma)
double t[] = new double[5]; //declaración y creación de la tabla con longitud 5

for (int i = 0; i < 5; i++) { //recorremos para leer los valores
    System.out.print("Introduzca un número: ");
    t[i] = sc.nextDouble();
}

System.out.println(Arrays.toString(t)); //muestra t
```

## Actividad resuelta 5.3

Escribir una aplicación que solicite al usuario cuántos números desea introducir. A continuación, introducir por teclado esa cantidad de números enteros, y por último, mostrar en el orden inverso al introducido.

### Solución

```
/* Primero leeremos la cantidad de números a introducir. Con esta información
 * creamos una tabla de la longitud adecuada para albergar todos los datos que se
 * introducirán por teclado. Por último, recorreremos la tabla, pero comenzando
 * en el último elemento y finalizando en el primero, con lo que conseguimos
 * mostrarlos en orden inverso. */

Scanner sc = new Scanner(System.in);
System.out.println("Cuántos números desea introducir: ");
int cuantosNumeros = sc.nextInt();

int t[] = new int[cuantosNumeros]; //tabla con la longitud adecuada

for (int i = 0; i < t.length; i++) { //recorremos desde 0 hasta t.length-1
    System.out.print("Introduzca un número: ");
    t[i] = sc.nextInt();
}

System.out.println("Los números en orden inverso son: ");
for (int i = t.length - 1; i >= 0; i--) { // recorremos en orden inverso
    System.out.println(t[i]);
}
//En este caso no podemos utilizar Arrays.toString() para mostrar la tabla
```

## Actividad resuelta 5.4

Diseñar la función: `int maximo(int t[])`, que devuelva el máximo valor contenido en la tabla `t`.

### Solución

```
static int maximo(int t[]) {
    int max = t[0]; // el primer elemento será, en principio, el máximo.
    //Suponemos que la tabla siempre tendrá al menos un elemento

    for (int e : t) { //recorremos para buscar un elemento mayor que max
        if (e > max) { // si e (t[i]) es mayor que max, es el nuevo máximo
            max = e;
        }
    }
    return (max);
}
```

## 5.7.5. Ordenación

Ordenar una tabla consiste en cambiar de posición los datos que contiene para que, en conjunto, resulten ordenados. La clase `Arrays` permite ordenar tablas mediante el método:

- `static void sort(tipo t[])`: ordena los elementos de la tabla `t` de forma creciente. El método se encuentra sobrecargado para cualquier tipo primitivo; de ahí que `tipo` pueda ser `int`, `double`, etcétera.

Veamos cómo ordenar una tabla:

```
int edad = {85, 19, 3, 23, 7}; //tabla desordenada
Arrays.sort(edad); //ordena la tabla. Ahora edad = [3, 7, 19, 23, 85]
```

### Argot técnico

Buscar en una tabla ordenada es una operación muy rápida; por el contrario, hacerlo en una tabla sin ordenar requiere de mucho tiempo. Sin embargo, el proceso de ordenación es muy largo. Por ello, antes de ordenar una tabla hay que plantearse si realmente es necesario, y si compensa hacerlo para que las búsquedas sean más rápidas. Solo merecerá la pena ordenar una tabla si vamos a realizar muchas búsquedas en ella.



## Actividad resuelta 5.5

Escribir la función `int[] rellenaPares(int longitud, int fin)`, que crea y devuelve una tabla ordenada de la longitud especificada, que se encuentra rellena con números pares aleatorios comprendidos en el rango desde 2 hasta `fin` (inclusive).

**Solución**

```

static int[] rellenaPares(int longitud, int fin) {
    //Creamos la tabla con la longitud adecuada
    int pares[] = new int[longitud];

    int i = 0; //indica con que elemento de la tabla estamos trabajando

    while (i < pares.length) { //terminaremos de llenar la tabla cuando el
        //número de pares sea igual que la longitud de la tabla
        int num = (int)(Math.random()*fin + 1);
        if (num % 2 == 0) { // si es par
            pares[i] = num; // lo guardamos en el elemento i
            i++; //incrementamos el indicador
        }
    }
    //ahora nos falta ordenar la tabla
    Arrays.sort(pares);
    return (pares);
}

```

 **5.7.6. Búsqueda**

Consiste en averiguar si entre los elementos de una tabla se encuentra, y en qué posición, un valor determinado llamado *clave de búsqueda*. El algoritmo de búsqueda depende de si la tabla está o no ordenada.

Una búsqueda en una tabla ordenada siempre es más rápida que buscar en la misma tabla con sus elementos no ordenados.

 **Búsqueda en una tabla no ordenada**

Se denomina *búsqueda secuencial* y consiste en un recorrido de la tabla donde se comprueban los valores de los elementos. El proceso finalizará cuando encontremos la clave de búsqueda o cuando no existan más elementos donde buscar. Dicho de otro modo, mientras no encontremos el valor buscado o el final de la tabla, hemos de continuar con la búsqueda. El siguiente algoritmo busca *claveBusqueda* en una tabla *t* no ordenada.

```

/* búsqueda secuencial */
int indiceBusqueda = 0; //índice que usamos para recorrer la tabla
while (indiceBusqueda < t.length && //no es el último elemento
      t[indiceBusqueda] != claveBusqueda) { //y no encontrado
    indiceBusqueda++; //incrementamos el índice de búsqueda
}
if (indiceBusqueda < t.length) {
    ... //claveBusqueda se encuentra en la posición indiceBusqueda
} else { //el índice se ha salido de rango
    ... //no encontrado
}

```

Cuando salimos del bucle `while` es por dos motivos: o bien hemos encontrado el elemento buscado o, por el contrario, no lo hemos encontrado y no hay más elementos donde buscar.

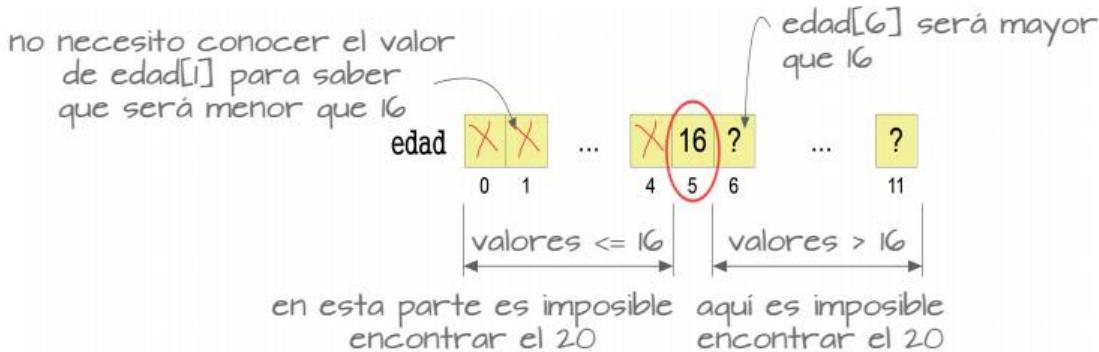
### Actividad propuesta 5.4

Escribe la función: `int buscar(int t[], int clave)`, que busca de forma secuencial en la tabla `t` el valor `clave`. En caso de encontrarlo, devuelve en qué posición lo encuentra; y en caso contrario, devolverá `-1`.

## Búsqueda en una tabla ordenada

Aprovechamos que la posición de los valores nos proporciona información extra. Supongamos que en la tabla `edad`, con una longitud de 12 y ordenada de forma creciente, buscamos si alguien tiene 20 años. Y sabemos que `edad[5]` es `16`. Sin necesidad de conocer más valores de la tabla, deducimos que:

- Los valores de `edad[0], ..., edad[4]` serán menores o iguales que 16, y aquí es imposible encontrar el 20.
- `edad[5]` vale 16.
- En caso de encontrarse, el valor 20 estará a partir de `edad[6]`.



**Figura 5.12.** Búsqueda dicotómica. El elemento central de una tabla proporciona información extra de dónde buscar.

En nuestro ejemplo (Figura 5.12) podemos descartar la primera mitad de la tabla, ya que al estar ordenada, es imposible encontrar el valor 20. En caso de existir, estará en la segunda mitad. Este algoritmo se repetirá sucesivas veces, siempre en la mitad donde sea posible encontrar el valor. Este comportamiento hace que sea la forma más eficiente de buscar.

Esta propiedad de las tablas ordenadas se aprovecha en el algoritmo de **búsqueda dicotómica** —también llamado *búsqueda binaria*—, que comprueba si la clave de búsqueda se encuentra en el elemento central de la tabla. Con esta información sabe si debe seguir buscando en la primera o en la segunda mitad de la tabla. El proceso se repite con la mitad, donde es posible encontrar la clave de búsqueda, que se subdivide de nuevo en dos partes. El algoritmo continúa hasta encontrar la clave de búsqueda o hasta que no existan más elementos donde buscar.

Podemos escribir nuestro propio algoritmo de búsqueda dicotómica, pero no es necesario, ya que se encuentra implementada en la clase `Arrays`.

- `static int binarySearch(tipo t[], tipo claveBusqueda)`: busca de forma dicotómica en la tabla `t` (que supone ordenada) el elemento con valor `claveBusqueda`. Devuelve el índice donde se encuentra la primera ocurrencia del elemento buscado o un valor negativo en caso contrario.

Veamos un ejemplo: deseamos buscar en la tabla ordenada `precios`, si existe y en qué posición está algún producto de 19,95 euros.

```
int pos = Arrays.binarySearch(precios, 19.95);
if (pos >= 0) {
    System.out.println("Encontrado en el índice: " + pos);
} else {
    System.out.println("Lo sentimos, no se ha encontrado.");
}
```

### Argot técnico



Cuando el elemento que buscar no se encuentra, el valor negativo devuelto tiene un significado especial: informa de la posición donde tendría que colocarse el elemento buscado para que la tabla continúe ordenada. El índice de inserción se calcula:

```
indiceInsercion = -pos - 1;
```

siendo `pos` el valor negativo devuelto por `binarySearch()`. Veamos un ejemplo:

```
int a[] = {2, 4, 5, 6, 9};
int pos = Arrays.binarySearch(a, 3); //pos vale -2
int indiceInsercion = -pos - 1; //vale 1
```

Es decir, si insertamos el valor buscado (3) en la tabla, debemos hacerlo en el índice 1 para que la tabla continúe ordenada.

### Actividad resuelta 5.6

Definir una función que tome como parámetros dos tablas, la primera con los 6 números de una apuesta de la primitiva, y la segunda (ordenada) con los 6 números de la combinación ganadora. La función devolverá el número de aciertos.

#### Solución

```
//Devuelve el número de coincidencias entre los elementos de las tablas
static int primitiva(int apuesta[], int[] combinacionGanadora) {
    int aciertos = 0; //contador de aciertos

    for (int a : apuesta) { // recorremos la tabla de apuesta
        //aprovechamos que la tabla con la combinación está ordenada
        if (Arrays.binarySearch(combinacionGanadora, a) >= 0) { //si está
            aciertos++; //hemos acertado un número más
        }
    }
    return (aciertos);
}
```

El método anterior realiza la búsqueda en toda la tabla, pero si solo interesa buscar en un subconjunto de elementos, disponemos de:

- `static int binarySearch(tipo t[], int desde, int hasta, tipo clave-Búsqueda)`: solo busca en los elementos comprendidos entre los índices `desde` y `hasta`, sin incluir en la búsqueda este último.

### Argot técnico



En los métodos de distintas clases de la API, cuando se describe un rango mediante dos índices, `desde` y `hasta`, es habitual que se incluya en el rango el valor `desde` y se excluya `hasta`.

## 5.7.7. Copia

El procedimiento para realizar manualmente la copia exacta de una tabla consiste en:

1. Crear una nueva tabla, que llamaremos `destino` o `copia`, del mismo tipo y longitud que la tabla original.
2. Recorrer la tabla original, copiando el valor de cada elemento en su lugar correspondiente en la tabla destino.

Sin embargo, `Arrays` proporciona esta funcionalidad mediante:

- `static tipo[] copyOf(tipo origen[], int longitud)`: construye y devuelve una copia de `origen` con la longitud especificada. Si la longitud de la nueva tabla es menor que la de la original, solo se copian los elementos que caben. En caso contrario, los elementos extras se inicializan por defecto. Este método, como la mayoría de los métodos de `Arrays`, está sobrecargado para poder trabajar con todos los tipos.

Veamos un ejemplo:

```
int t[] = {1, 2, 1, 6, 23}; //tabla origen
int a[], b[]; // tablas destino
a = Arrays.copyOf(t, 3); //a = [1, 2, 1]
b = Arrays.copyOf(t, 10); //b = [1, 2, 1, 6, 23, 0, 0, 0, 0, 0]
```

Existe otro método que también realiza una copia de una tabla, pero en este caso de un rango de elementos:

- `static tipo[] copyOfRange(tipo origen[], int desde, int hasta)`: crea y devuelve una tabla donde se han copiado los elementos de `origen` comprendidos entre los índices `desde` y `hasta`, sin incluir este último.

Un ejemplo:

```
int t[] = {7, 5, 3, 1, 0, -2};
int a[] = Arrays.copyOfRange(t, 1, 4); //a = [5, 3, 1]
```

que realiza una copia desde los índices 1 al 3 (el anterior al 4).

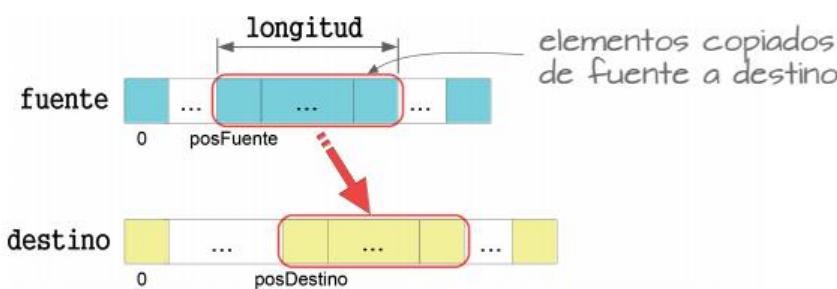
Otro método disponible es `arraycopy()` de la clase `System`, que copia elementos consecutivos entre dos tablas. La diferencia entre `arraycopy()` y `copyOfRange()` es que el primero no crea ninguna tabla, ambas tablas deben estar creadas previamente. Su sintaxis es:

- `void arraycopy(Object tablaOrigen, int posOrigen, Object tablaDestino, int posDestino, int longitud)`: copia en la `tablaDestino`, a partir del índice `posDestino`, los datos de la `tablaOrigen`, comenzando en el índice `posOrigen`. El parámetro `longitud` especifica el número de elementos que se copiarán entre ambas tablas. Hay que tener precaución, ya que los valores de los elementos afectados por la copia de la tabla destino se perderán. Véase la Figura 5.13.

### Argot técnico



`Object` es una forma de llamar en Java a cualquier cosa, incluida las tablas. Por ahora no estamos trabajando con clases, pero pronto entraremos de lleno en el maravilloso mundo de la programación orientada a objetos.



**Figura 5.13.** Proceso de copia que realiza `copyarray()`. Los elementos copiados pueden moverse desde su posición en la tabla fuente (u origen) a cualquier otra posición en la tabla destino. Ambas tablas pueden ser de distinta longitud, aunque siempre hay que estar seguro de que no copiaremos en elementos fuera de rango, lo que produciría un error.

### 5.7.8. Inserción

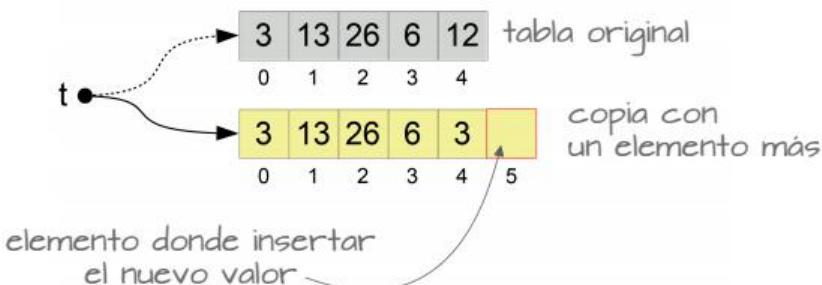
La forma de añadir un nuevo valor a una tabla depende de si está o no ordenada. Si el orden no importa, basta con incrementar la longitud de la tabla e insertar el nuevo dato en el último elemento. Y cuando la tabla está ordenada, hemos de insertar el nuevo dato de forma que todos los valores sigan ordenados.

### Inserción no ordenada

Veamos el algoritmo para insertar el valor `nuevo`, en un elemento que añadimos al final de la tabla `t`. Hay que destacar que la longitud de la tabla no se modifica; lo que realmente ocurre es que se crea una segunda tabla (copia de la tabla original) en la que hemos aumentado la longitud en uno. La nueva tabla se referencia con la misma variable `t`, dando la sensación de que la hemos modificado. La tabla original, al quedar sin referencia,

queda a merced del recolector de basura. La Figura 5.14 representa lo que ocurre en el siguiente código:

```
t = Arrays.copyOf(t, t.length + 1); //la copia incrementa la longitud
t[t.length-1] = nuevo;
```



**Figura 5.14.** Inicialmente la tabla original estaba referenciada por `t`. Tras ejecutar `Arrays.copyOf()`, `t` se modifica y pasa a referenciar a la nueva tabla, que es una copia con una longitud incrementada en uno. Ahora disponemos de un nuevo elemento para añadir un dato más.

## Actividad resuelta 5.7

Implementar la función: `int[] sinRepetidos(int t[])`, que construye y devuelve una tabla de la longitud apropiada, con los elementos de `t`, donde se han eliminado los datos repetidos.

### Solución

```
/* Vamos a crear una tabla con longitud inicial de 0, a la que llamaremos
 * temporal. Recorreremos la tabla t comprobando que sus elementos no se
 * encuentra en la tabla temporal (aprovecharemos el método buscar() creado
 * en la actividad propuesta 5.1). Si el elemento no está en temporal, lo
 * insertaremos. */
static int[] sinRepetidos(int[] t) {
    int temporal[] = new int[0]; //Creamos con longitud 0

    for (int elemento : t) {
        if (buscar(temporal, elemento) == -1) { //Si no está: insertamos
            //algoritmo de inserción
            temporal = Arrays.copyOf(temporal, temporal.length + 1);
            temporal[temporal.length - 1] = elemento; //Añadimos al final
        }
    }
    return temporal;
}
```

## Actividad resuelta 5.8

Leer y almacenar  $n$  números enteros en una tabla, a partir de la que se construirán otras dos tablas con los elementos con valores pares e impares de la primera, respectivamente. Las tablas pares e impares deben mostrarse ordenadas.

### Solución

```
/* Como las tablas con los números pares e impares tienen que estar ordenadas,
 * lo que haremos será ordenar los datos de entrada. Que recorremos y, según
 * sea par o impar, se insertará en la tabla correspondiente.*/
Scanner sc = new Scanner(System.in);
int datos[]; //tabla para los datos iniciales
//Creamos las tablas par e impar, inicialmente de longitud 0:
int par[] = new int[0];
int impar[] = new int[0];

System.out.print("Escriba n: ");
int n = sc.nextInt(); //n es el número de datos a leer
datos = new int[n]; //Creamos la tabla de longitud n

//leemos del teclado los valores de la tabla
for (int i = 0; i < datos.length; i++) {
    System.out.print("Introduzca un número: ");
    datos[i] = sc.nextInt();
}

//recorremos los datos para clasificarlos
for (int numero: datos) {
    //Al estar la tabla con todos los datos ordenadas, los elementos
    //se insertarán siempre al final de la tabla par o impar.
    if (numero % 2 == 0) { //si número es par
        par = Arrays.copyOf(par, par.length+1); //incremento la longitud de par
        par[par.length-1] = numero; //guardo el número en el último elemento
    } else {
        impar = Arrays.copyOf(impar, impar.length+1); //igual con la tabla impar
        impar[impar.length-1] = numero;
    }
}

System.out.println("Pares: " + Arrays.toString(par));
System.out.println("Impares: " + Arrays.toString(impar));
```

## Inserción ordenada

La inserción ordenada consiste en añadir un nuevo elemento en la tabla en la posición adecuada para que la tabla continúe ordenada. Primeramente, buscaremos el lugar que le correspondería al nuevo valor en la tabla; a este índice le llamaremos `indiceInsercion`. A continuación, crearemos una nueva tabla, que llamaremos `copia`, con un elemento extra.

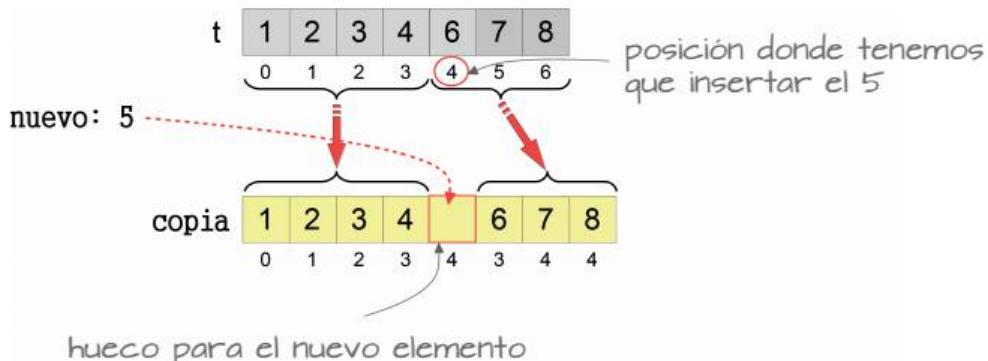
Ahora hemos de copiar los elementos de la tabla `original` a la tabla `copia`, teniendo la precaución de no utilizar el elemento situado en `indiceInsercion`, que es un hueco que está reservado para el nuevo valor. Es decir, todos los elementos cuyos índices son anteriores a `indiceInsercion` se copian en la misma posición, y los posteriores, se copian desplazados un elemento hacia el final de la tabla. Con esto conseguimos que, tras insertar el nuevo valor en el elemento marcado por `indiceInsercion`, la tabla se mantenga ordenada (véase la Figura 5.15).

Finalmente, la copia será referenciada por la misma variable que referenciaba la tabla original, dando la sensación de que la tabla ha crecido. Veamos un ejemplo:

```

int t[] = {1, 2, 3, 4, 6, 7, 8};
int nuevo = 5;
int indiceInsercion = Arrays.binarySearch(t, nuevo);
//si indiceInsercion >= 0, el nuevo elemento (que está repetido) se inserta en
//el lugar en que ya estaba, desplazando al original. Si por el contrario:
if (indiceInsercion < 0) { //si no lo encuentra
    //calcula donde debería estar
    indiceInsercion = -indiceInsercion - 1;
}
int copia[] = new int[t.length + 1]; //nueva tabla con longitud+1
//copiamos los elementos antes del "hueco"
System.arraycopy(t, 0, copia, 0, indiceInsercion);
//copiamos desplazados los elementos tras el "hueco"
System.arraycopy(t, indiceInsercion,
copia, indiceInsercion+1, t.length - indiceInsercion);
copia[indiceInsercion] = nuevo; //asignamos el nuevo elemento
t = copia; //t referencia la nueva tabla
System.out.println(Arrays.toString(t)); //mostramos

```



**Figura 5.15.** Momento en el que hemos creado el hueco para el nuevo elemento y hemos copiado los datos. Solo queda insertar el nuevo elemento (5) en el hueco y referenciar la tabla copia con t, dando la sensación de que la tabla ha incrementado su longitud. Es importante que, tras todas las operaciones, la tabla t continúe estando ordenada.

## Actividad propuesta 5.5

Escribe en una función el comportamiento de la inserción ordenada.

## Actividad resuelta 5.9

Diseñar una aplicación para gestionar un campeonato de programación, donde se introduce la puntuación (enteros) obtenidos por 5 programadores, conforme van terminando su prueba. La aplicación debe mostrar las puntuaciones ordenadas de los 5 participantes. En ocasiones, cuando finalizan los 5 participantes anteriores, se suman al campeonato programadores de exhibición, cuyos puntos se incluyen con el resto. La forma de especificar

que no intervienen más programadores de exhibición es introducir como puntuación un -1. La aplicación debe mostrar, finalmente, los puntos ordenados de todos los participantes.

### Solución

```
/* Leeremos las puntuaciones en el orden en el que terminen los participantes y
 * las ordenaremos. A continuación, realizaremos una inserción ordenada (por
 * cada programador de exhibición). Una mala idea sería insertar al final la
 * puntuación de los programadores de exhibición y volver a ordenar, ya que
 * esto es muy costoso en tiempo. Es más eficiente una inserción ordenada. */
Scanner sc = new Scanner(System.in);
int puntos[] = new int[5]; //inicialmente intervienen 5 programadores

for (int i = 0; i < 5; i++) {
    System.out.print("Puntos programador (" + (i + 1) + "): ");
    puntos[i] = sc.nextInt(); //leemos los datos, que no están ordenados
}

Arrays.sort(puntos); //ordenamos
System.out.println("Puntuación: " + Arrays.toString(puntos));

System.out.print("Puntos del programador de exhibición: ");
int puntosProgExh = sc.nextInt(); //puntuación del prog. de exhibición
while (puntosProgExh != -1) {
    int pos = Arrays.binarySearch(puntos, puntosProgExh); //buscamos
    int indiceInsercion; //donde insertar para que la tabla siga ordenada
    if (pos < 0) {
        indiceInsercion = -pos - 1; //índice para que la tabla esté ordenada
    } else {
        indiceInsercion = pos; //puntuación repetida, ya está en la tabla
    }

    int copia[] = new int[puntos.length + 1]; //nueva tabla con longitud+1
    //copiamos los elementos antes del "hueco"
    System.arraycopy(puntos, 0, copia, 0, indiceInsercion);
    //copiamos desplazados los elementos tras el "hueco"
    System.arraycopy(puntos, indiceInsercion,
                     copia, indiceInsercion + 1, puntos.length - indiceInsercion);

    copia[indiceInsercion] = puntosProgExh; //asignamos el nuevo elemento
    puntos = copia; //puntos referencia la nueva tabla

    System.out.print("Puntos del programador de exhibición: ");
    puntosProgExh = sc.nextInt(); //puntuación del prog. de exhibición
}

System.out.println("Puntuación final: " + Arrays.toString(puntos));
```

## ■ ■ ■ 5.7.9. Eliminación

Esta operación consiste en borrar un elemento de la tabla, por lo que después de una eliminación la longitud de la tabla decrece. Antes de eliminar un elemento de una tabla, siempre tendremos que buscarlo para conocer en qué índice se encuentra. Una vez localizado, la operación dependerá del tipo de tabla con la que estemos trabajando.

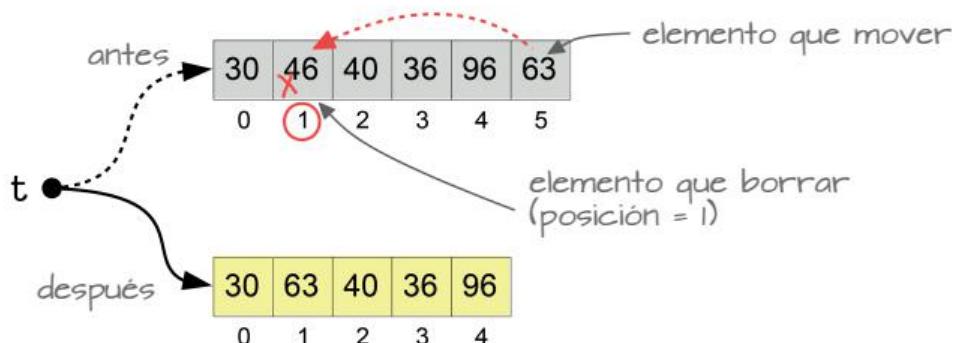
## ■ ■ ■ Tabla no ordenada

Para eliminar un elemento en una tabla, después de buscarlo, lo sustituimos por el último dato de la tabla —que ahora estará repetido—. A continuación, creamos una copia de la tabla con los mismos datos, pero disminuyendo su longitud, lo que provoca que perdamos el último elemento, que es el que estaba repetido.

Veamos el algoritmo donde `t []` es la tabla con los datos e `indiceBorrado` contendrá, si existe, el índice del elemento que deseamos eliminar, que se almacena en la variable `aBorrar`:

```
... //algoritmo de búsqueda, que devuelve el índice del elemento a borrar si
//existe, o -1 si no existe.
if (indiceBorrado != -1) { //encontrado
    t[indiceBorrado] = t[t.length - 1]; //copia el último en indiceBorrado
    t = Arrays.copyOf(t, t.length - 1); //disminuimos la longitud de t
    System.out.println(Arrays.toString(t)); //mostramos
} else {
    ... //no podemos borrar nada, ya que no lo hemos encontrado
}
```

`aBorrar: 46`



**Figura 5.16.** Todas las operaciones de eliminación comienzan localizando el elemento que se va a borrar. Despues se han de recolocar el resto de elementos para que produzcan el efecto de que el elemento en cuestión ha desaparecido. Finalmente, redimensionamos a una tabla más pequeña.

### Actividad resuelta 5.10

Escribir la función:

```
int[] eliminarMayores(int t[], int valor)
```

que crea y devuelve una copia de la tabla `t` donde se han eliminado todos los elementos que son mayores que `valor`.

#### Solución

```
static int[] sinMayores(int t[], int valor) {
    int copia[] = Arrays.copyOf(t, t.length); //copia es un clon de t
    int i=0;
```

```

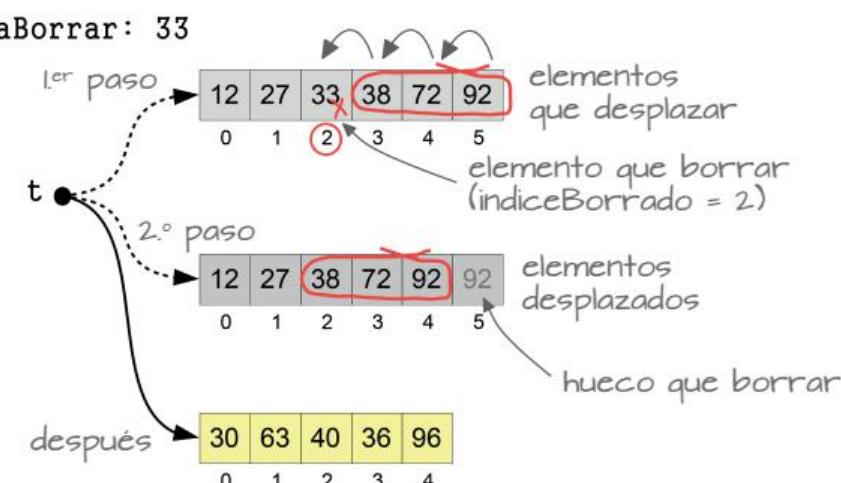
while (i<copia.length) { //recorremos copia
    if (copia[i] > valor) {
        //hay que eliminar copia[i]:
        copia[i] = copia[copia.length-1]; //copiamos el último en copia[i]
        //y decrementamos la longitud de copia en 1. Elimina el último.
        copia = Arrays.copyOf(copia, copia.length-1);
        //ahora tendremos que volver a comprobar copia[i]. No modificamos i
    } else {
        i++; //copia[i] se queda en la tabla, comprobaremos copia[i+1]
    }
}

return copia;
}

```

## Tablas ordenadas

El primer paso es buscar el elemento que se va a borrar (variable `aBorrar`). Al estar la tabla ordenada podemos utilizar la búsqueda dicotómica. Este algoritmo busca el índice (variable `indiceBorrado`) que utilizaremos para determinar el elemento que eliminar. En tablas ordenadas, al eliminar un elemento tenemos que seguir manteniendo los demás valores contiguos y en el mismo orden. Para ello, tenemos que desplazar los valores que siguen a `indiceBorrado` una posición hacia el principio. Con esta técnica sobrescribimos el valor que borrar y obtenemos un hueco libre al final de la tabla, que desaparecerá al disminuir su longitud.



**Figura 5.17.** El borrado en una tabla ordenada implica un primer paso donde se localiza el elemento que borrar, un segundo paso donde se desplazan los elementos a la derecha del elemento que nos interesa y, finalmente, el redimensionado de la tabla.

Veamos a modo de ejemplo cómo eliminar un elemento que se solicita por teclado:

```

int t[] = {12, 27, 33, 38, 72, 92};
int aBorrar = new Scanner(System.in).nextInt();
//usamos el algoritmo de búsqueda dicotómica
int indiceBorrado = Arrays.binarySearch(t, aBorrar);
if (indiceBorrado >= 0) {

```

```
//desplazamos los elementos posteriores a indiceBorrado
System.arraycopy(t, indiceBorrado + 1,
t, indiceBorrado, t.length - indiceBorrado - 1);
t = Arrays.copyOf(t, t.length - 1); //disminuimos la longitud
System.out.println(Arrays.toString(t)); //mostramos
} else {
... //no podemos borrar nada, ya que no lo hemos encontrado
}
```

## Actividad propuesta 5.6

Crea una función que realice el borrado de un elemento de una tabla ordenada.

## Actividad propuesta 5.7

El «número de la suerte» de una persona puede calcularse a partir de sus números favoritos. De entre estos, se seleccionan dos diferentes al azar, que se eliminarán de la lista, pero en su lugar se añade la media aritmética de los dos eliminados a la lista de números favoritos. El proceso se repite hasta que solo quede un número, que resultará el número de la suerte para esa persona. Para calcular bien el número de la suerte es imprescindible que la lista de números se encuentre siempre ordenada.

Escribe una aplicación que solicite al usuario sus números favoritos y calcula su número de la suerte.

### 5.7.10. Comparación de dos tablas

El contenido de dos tablas no se puede comparar mediante el operador `==`, ya que este operador no compara los elementos de las tablas, sino sus referencias. Por ejemplo:

```
int t1[] = {7, 9, 20};
int t2[] = {7, 9, 20}; //t1 y t2 tienen los mismos elementos
System.out.println(t1 == t2); //sin embargo muestra false
```

ya que `t1` y `t2` tienen los mismos elementos, pero distintas referencias (están ubicadas en distintas zonas de la memoria).

Dos tablas se consideran iguales si contienen los mismos elementos en el mismo orden. Para comparar dos tablas disponemos del método de `Arrays`.

- `static boolean equals(tipo a[], tipo b[]):` compara las tablas `a` y `b`, elemento a elemento. En el caso de que sean iguales devuelve `true`, y en caso contrario, `false`.

Veamos cómo comparar las dos tablas anteriores:

```
System.out.println(Arrays.equals(t1, t2)); //muestra true
```

## Actividad propuesta 5.8

Comprueba qué produce la comparación con el operador `==` de dos tablas del mismo tipo, la misma longitud y los mismos valores.

## Actividad resuelta 5.11

Desarrollar el juego «la cámara secreta», que consiste en abrir una cámara mediante su combinación secreta, que está formado por una combinación de dígitos del uno al cinco. El jugador especificará cuál es la longitud de la combinación; a mayor longitud, mayor será la dificultad del juego. La aplicación genera, de forma aleatoria, una combinación secreta que el usuario tendrá que acertar. En cada intento se muestra como pista, para cada dígito de la combinación introducido por el jugador, si es mayor, menor o igual que el correspondiente en la combinación secreta.

### Solución

```

/*
 * El juego consiste en acertar la combinación secreta, que se genera de forma
 * aleatoria.
 */
public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);
    System.out.println("Longitud de la combinación secreta: ");
    int longitud = sc.nextInt();
    int combSecreta[] = new int[longitud]; //combinación secreta
    int combJugador[] = new int [longitud]; //combinación del jugador

    //generamos aleatoriamente la combinación secreta:
    generaCombinacion(combSecreta);
    //esto es trampa: mostramos la combinación secreta para facilitar
    System.out.println(Arrays.toString(combSecreta));
    System.out.println("Escriba una combinación:");
    leeTabla(combJugador);

    while (!Arrays.equals(combSecreta, combJugador)) { //no sean iguales
        muestraPistas(combSecreta, combJugador); //mostramos las pistas
        System.out.println("Escriba una combinación: ");
        leeTabla(combJugador); //volvemos a pedir otra combinación
    }
    //Salir del while significa que hemos acertado la combinación secreta:
    System.out.println(";La cámara está abierta!");
}

//Esta función inicializa los valores de la tabla t con valores aleatorios.
//La constante MAX determina el valor máximo que se asigna a un elemento,
//estando comprendido en el rango 1..MAX
static void generaCombinacion(int t[]) {
    final int MAX = 5; //rango 1..MAX
    for (int i = 0; i < t.length; i++) {
        t[i] = (int) (Math.random()*MAX+1); //número aleatorio de 1 a MAX
        //t referencia a la tabla combSecreta del programa principal. Por este
        //motivo asignar un valor a t[i] es lo mismo que hacerlo a
        //combSecreta[i]
    }
}

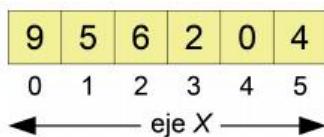
//Recorre t y asigna a cada elemento un valor leído desde el teclado
static void leeTabla(int t[]) {
    Scanner sc = new Scanner(System.in);
    for (int i = 0; i < t.length; i++) { //recorremos para leer
        t[i] = sc.nextInt();
    }
}

```

```
//Recorre las dos tablas, secret y jug, e indica para cada elemento de la
//combinación introducida por el usuario si es mayor, menor o igual que el
//equivalente en la combinación secreta
static void muestraPistas(int secret[], int jug[]) {
    System.out.println("Pistas:");
    for (int i = 0; i < jug.length; i++) { //recorremos ambas tablas
        System.out.print(jug[i]);
        if (secret[i] > jug[i]) { //comparamos el i-ésimo elemento de ambas
            System.out.println(" mayor");
        } else if (secret[i] < jug[i]) {
            System.out.println(" menor");
        } else {
            System.out.println(" igual");
        }
    }
}
```

## ■ 5.8. Tablas $n$ -dimensionales

Hasta el momento las tablas que hemos utilizado han sido unidimensionales: solo tienen longitud; dicho de otra forma, los elementos solo se extienden a lo largo de un eje (el eje  $X$ ), y basta con un índice para recorrerlas.



**Figura. 5.18.** El eje  $X$ , o eje de abscisa, es el eje horizontal. Hay que entender que esto es una representación, y lo habitual es que nos figuremos que las tablas unidimensionales se expanden horizontalmente.

Pero puede ocurrir que nuestros datos se refieran a entidades que se caracterizan por más de una propiedad, de las cuales alguna o algunas sirven para identificarlo. En este caso, no nos basta con un solo índice para describirlas.

### ■ 5.8.1. Tablas bidimensionales

Podemos ampliar el concepto de tabla haciendo que los elementos se extiendan en dos dimensiones, utilizando los ejes  $X$  e  $Y$ . Ahora la tabla posee longitud y anchura. Para identificar cada elemento de una tabla unidimensional hemos utilizado un índice; para las tablas bidimensionales, compuestas por filas y columnas, se necesita un par de índices  $[x][y]$ . Una tabla bidimensional recibe el nombre de *matriz*, aunque a diferencia de las matemáticas, en Java se numeran comenzando por 0.

La declaración de una tabla bidimensional se hace de la forma:

```
tipo nombreTabla[][];
```

A continuación, creamos la tabla, indicando la longitud de cada dimensión. Veamos cómo crear la tabla `datos` correspondiente a la Figura 5.19:

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

y con ello, estamos reservando espacio en la memoria para  $(5 \times 5) = 25$  elementos.

	0	1	2	3	4	
0	2	4	6	12	0	
1	2	-11	4	7	86	
2	0	1	6	5	3	
3	1	93	6	-2	0	
4	9	71	23	2	8	

**Figura 5.19.** Matriz de  $5 \times 5$  elementos. Ahora la identificación de cada elemento viene dada por el índice del eje X y el índice del eje Y.

Los algoritmos que utilizan matrices requieren dos bucles anidados. Un bucle se encarga del índice para la dimensión X y el otro para el índice del eje Y. Veamos un ejemplo para introducir por teclado la matriz `datos`:

```
for (i = 0; i < 5; i++) { //eje X
    for (j = 0; j < 5; j++) { //eje Y
        datos[i][j] = sc.nextInt(); //leemos el elemento [i][j]
    }
}
```

Para mostrar una tabla bidimensional podemos usar dos bucles anidados como los anteriores o bien utilizar el método estático `Arrays.deepToString()`. Por ejemplo, para mostrar la tabla `datos`,

```
System.out.println(Arrays.deepToString(datos));
```

## Actividad resuelta 5.12

Crear una tabla bidimensional de longitud  $5 \times 5$  y rellenarla de la siguiente forma: el elemento de la posición  $[n][m]$  debe contener el valor  $10 \times n + m$ . Después se debe mostrar su contenido.

### Solución

```
int t[][]; // declaramos t como una tabla bidimensional
t = new int[5][5]; // creamos la tabla de 5x5

for (int i = 0; i < 5; i++) { // utilizamos i para la primera dimensión
    for (int j = 0; j < 5; j++) { // y j para la segunda dimensión
        t[i][j] = 10*i + j;
    }
}
```

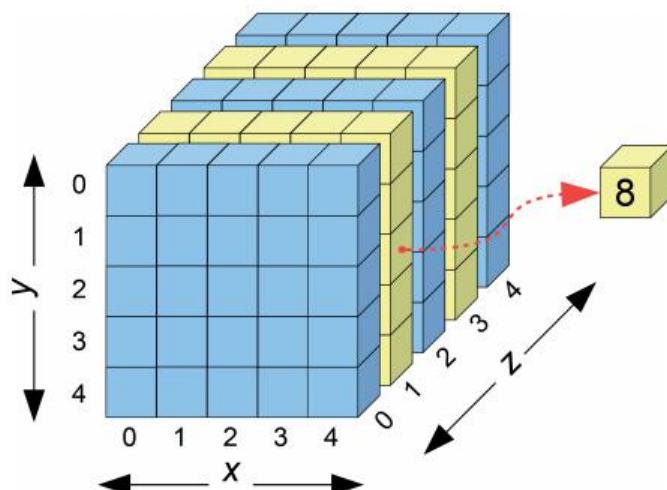
```

System.out.println(Arrays.deepToString(t)); //mostramos
//otra forma de mostrar es hacerlo recorriendo nosotros la matriz.
//Una matriz es un conjunto de filas (tabla unidimensional). Y cada fila
//está compuesta por una serie de elementos.
for (int fila[] : t) {
    for (int columna: fila) {
        System.out.print(columna + " ");
    }
    System.out.println();
}
}

```

## 5.8.2. Tablas tridimensionales

Añadiendo una nueva dimensión podemos crear tablas con anchura, altura y profundidad, es decir, tablas tridimensionales. Estas utilizan tres índices ( $[x][y][z]$ ) para identificar cada elemento que la componen.



**Figura 5.20.** Una tabla tridimensional se representa mediante un cubo. Se aprecia cómo se necesitan tres dimensiones (tres índices) para poder identificar cualquier elemento, que contiene un dato del tipo del que está declarada la tabla. En nuestro ejemplo, el índice  $[4][2][1]$  (que corresponde a  $x, y, z$ ) vale 8.

## 5.8.3. Tablas con más dimensiones

Dibujar o imaginar una tabla de más de tres dimensiones es algo complicado. Nosotros vivimos en un mundo 4-dimensional, con tres dimensiones para localizar cada objeto en el espacio y una cuarta dimensión, el tiempo, para ver la evolución de un objeto en movimiento. Pero más allá de nuestro mundo, es complicado representar, o siquiera imaginar, una tabla multidimensional. Un truco sencillo consiste en descomponer la tabla en otras más simples. Por ejemplo, una tabla de 5 dimensiones puede verse como una tabla tridimensional, donde en cada elemento de la tabla se almacena una tabla bidimensional. De los cinco índices necesarios para identificar los elementos de una tabla 5-dimensional, podemos utilizar los tres primeros en la tabla tridimensional y utilizar los otros dos índices para situarnos en la segunda tabla bidimensional.

Pero el hecho de que no podamos dibujarla ni imaginarla no significa que no se puedan manipular sus elementos.

Las tablas  $n$ -dimensionales son útiles para manejar la información atendiendo a criterios de clasificación. No es necesario que la información tenga una representación gráfica. Veamos un ejemplo: supongamos una máquina que procesa naranjas y necesitamos, por motivos de calidad, clasificarlas y llevar la cuenta del número de frutas recogidas de cada tipo, atendiendo a los criterios: diámetro, color, maduración, forma y peso.

Al utilizar cinco criterios, lo más apropiado para almacenar los datos es una matriz 5-dimensional, haciendo corresponder cada dimensión con un criterio de clasificación. Falta, para cada una de las dimensiones (criterios), formalizar una correspondencia entre los posibles valores reales de un criterio (color: naranja, amarillo o verde; nivel de maduración: madura o inmadura; etc.) con las longitudes de cada dimensión, que utilizaremos como índices. Una posible correspondencia puede ser la que se muestra en la Figura 5.21.

Diámetro	0	1	2	
	Pequeño, < 4 cm	Mediano, entre 4 y 8 cm	Grande, > 8 cm	
Color	0	1	2	
	Naranja	Amarillo	Verde	
Maduración	0	1	2	4
	Pasada	Óptima	Algo inmadura	Totalmente inmadura
Forma	0	1		
	Redondeada	Otra forma		
Peso	0	1	2	3
	<100 g	100-200 g	200-300 g	300-400 g
		4	5	
		400-500 g	>500 g	

**Figura 5.21.** Correspondencia entre los valores de cada dimensión y clasificaciones de distintos criterios de las naranjas.

Crearemos la variable `naranjas`, que será una matriz con 5 dimensiones, donde cada una de ellas representa un criterio: `naranjas [diámetro] [color] [maduración] [forma] [peso]`.

La declaración y creación de la variable es:

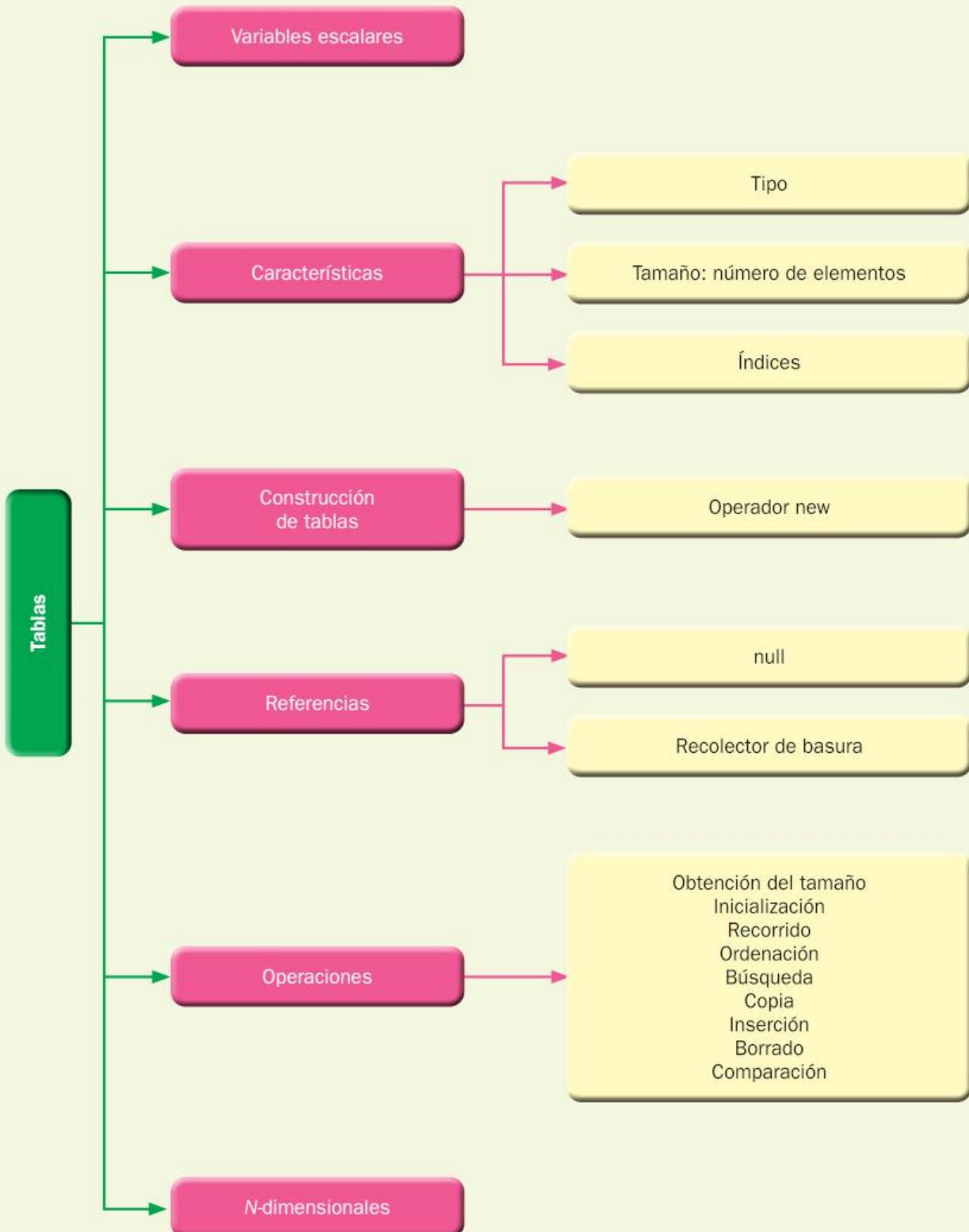
```
int naranjas[][][][];
naranjas = new int[3][3][5][2][6];
```

Si la máquina contabiliza 25 naranjas con:

- Diámetro de 11 cm: primer índice 2.
- De un color naranja intenso: segundo índice 0.
- En su punto óptimo de maduración: tercer índice 1.
- La forma es redonda: cuarto índice 0.
- Pesa 385 g: quinto índice 3.

Haremos la asignación:

```
naranjas[2][0][1][0][3] = 25;
```



## Actividades de comprobación

- 5.1.** Una tabla puede almacenar datos de distintos tipos, como por ejemplo enteros, booleanos, reales, etcétera:
- a) Cierto, las tablas siempre pueden almacenar datos de distintos tipos.
  - b) Falso, las tablas solo pueden almacenar datos de un único tipo.
  - c) Puede almacenar datos de distintos tipos siempre que estos sean numéricos.
  - d) Puede almacenar datos de distintos tipos siempre que la longitud de los datos sea idéntica.
- 5.2.** En Java, la numeración de los índices que determina la identificación de cada elemento de una tabla comienza en:
- a) Cero.
  - b) Uno.
  - c) Depende del tipo de dato de la tabla.
  - d) Es configurable por el usuario.
- 5.3.** Si en una tabla de 10 elementos utilizamos el elemento con índice 11 (que se encuentra fuera de rango):
- a) Al salir del rango de la longitud, Java redimensiona la tabla de forma automática.
  - b) No es posible y produce un error.
  - c) Las tablas tienen un comportamiento circular y utilizar el índice 11 es idéntico a utilizar el índice 1.
  - d) Ninguna de las anteriores respuestas es cierta.
- 5.4.** ¿Qué método de la clase Arrays permite realizar una búsqueda dicotómica en una tabla?
- a) `Arrays.search()`.
  - b) `Arrays.find()`.
  - c) `Arrays.binarySearch()`.
  - d) Cualquiera de los métodos anteriores realiza una búsqueda.
- 5.5.** Con respecto a las tablas, el operador `new`:
- a) Destruye, crea y redimensiona tablas.
  - b) Destruye y crea tablas.
  - c) Crea tablas.
  - d) Destruye las tablas.
- 5.6.** La forma de invocar al recolector de basura es:
- a) Mediante `System.garbageCollector()`.
  - b) Mediante el operador `new`.
  - c) Mediante `Arrays.garbageCollector()`.
  - d) Ninguna de las anteriores respuestas es correcta.
- 5.7.** La forma de conocer la longitud de una tabla `t` es mediante:
- a) `t.size`.
  - b) `t.elements`.
  - c) `t.length`.
  - d) `Arrays.size(t)`.

- 5.8.** La comparación del contenido (los elementos) de dos tablas se realiza utilizando:
- a) `Arrays.compare()`.
  - b) El operador `==`.
  - c) `Arrays.equals()`.
  - d) `Arrays.same()`.
- 5.9.** ¿Qué condición tiene que cumplir una tabla para que podamos realizar búsquedas dicotómicas en ella?
- a) Que esté ordenada.
  - b) Que esté ordenada y sea una tabla de enteros.
  - c) Que no esté ordenada.
  - d) No importa si la tabla está ordenada, lo realmente importante es que sea de algún tipo numérico.
- 5.10.** ¿Cuál es la principal diferencia entre `Arrays.copyOf()` y `System.arraycopy()`?
- a) No existe diferencia alguna, ambos métodos son idénticos.
  - b) `Arrays.copyOf()` copia mientras `System.arraycopy()` copia y compara.
  - c) `Arrays.copyOf()` copia entre tablas existentes mientras `System.arraycopy()` crea una nueva tabla y copia en ella.
  - d) `Arrays.copyOf()` crea una nueva tabla y copia en ella mientras `System.arraycopy()` solo copia entre tablas ya creadas.

## Actividades de aplicación

- 5.11.** Realiza la función: `int[] buscarTodos(int t[], int clave)`, que crea y devuelve una tabla con todos los índices de los elementos donde se encuentra la clave de búsqueda. En el caso de que `clave` no se encuentre en la tabla `t`, la función devolverá una tabla vacía.
- 5.12.** Escribe la función `void desordenar(int t[])`, que cambia de forma aleatoria los elementos contenidos en la tabla `t`. Si la tabla estuviera ordenada, dejaría de estarlo.
- 5.13.** Modifica la Actividad de aplicación 5.12 para que la función no modifique la tabla que se pasa como parámetro y, en su lugar, cree y devuelva una copia de la tabla donde se han desordenado los valores de los elementos.
- 5.14.** El ayuntamiento de tu localidad te ha encargado una aplicación que ayude a realizar encuestas estadísticas para conocer el nivel adquisitivo de los habitantes del municipio. Para ello, tendrás que preguntar el sueldo a cada persona encuestada. *A priori*, no conoces el número de encuestados. Para finalizar la entrada de datos, introduce un sueldo con valor `-1`.

Una vez terminada la entrada de datos, muestra la siguiente información:

- Todos los sueldos introducidos ordenados de forma decreciente.
- El sueldo máximo y mínimo.
- La media de los sueldos.

- 5.15.** Debes desarrollar una aplicación que ayude a gestionar las notas de un centro educativo. Los alumnos se organizan en grupos compuestos por 5 personas. Leer las notas (números enteros) del primer, segundo y tercer trimestre de un grupo. Debes mostrar al final la nota media del grupo en cada trimestre y la media del alumno que se encuentra en una posición dada (que el usuario introduce por teclado).
- 5.16.** En un juego de rol el mapa puede implementarse como una matriz donde las filas y las columnas representan lugares (lugar 0, lugar 1, lugar 2, etc.) que estarán conectados. Si desde el lugar X podemos ir hacia el lugar Y, entonces la matriz en la posición [x][y] valdrá cierto; en caso contrario, valdrá falso. Escribe una función que, dada una matriz que representa el mapa y dos lugares, indique si es posible viajar desde el primer lugar al segundo (directamente o pasando por lugares intermedios).
- 5.17.** Implementa la función: `int[] suma(int t[], int numElementos)`, que crea y devuelve una tabla con las sumas de los `numElementos` elementos consecutivos de `t`. Veamos un ejemplo, sea `t = [10, 1, 5, 8, 9, 2]`. Si los elementos de `t` se agrupan de 3 en 3, se harán las sumas:
- 10 + 1 + 5. Igual a 16.  
1 + 5 + 8. Igual a 14.  
5 + 8 + 9. Igual a 22.  
8 + 9 + 2. Igual a 19.
- Por lo tanto, la función devolverá una tabla con los resultados: [16, 14, 22, 19].
- 5.18.** Escribe un programa que solicite los elementos de una matriz de tamaño  $4 \times 4$ . La aplicación debe decidir si la matriz introducida corresponde a una matriz mágica, que es aquella donde la suma de los elementos de cualquier fila o de cualquier columna valen lo mismo.
- 5.19.** Diseña una aplicación para gestionar la llegada a meta de los participantes de una carrera. Cada uno de ellos dispone de un dorsal (un número entero) que los identifica. En la aplicación se introduce el número de dorsal de cada corredor cuando este llega a la meta. Para indicar que la carrera ha finalizado (han llegado todos los corredores a la meta), se introduce como dorsal el número  $-1$ .
- A continuación, la aplicación solicita información extra de los corredores. En primer lugar, se introducen los dorsales de todos los corredores menores de edad; para premiarlos por su esfuerzo se les avanza un puesto en el ranking general de la carrera, es decir, es como si hubieran adelantado al corredor que llevaban delante. En segundo lugar, se introducen los dorsales de los corredores que han dado positivo en el test antidopaje, lo que provoca su expulsión inmediata. Para finalizar, se introducen los dorsales de los corredores que no han pagado su inscripción en la carrera, lo que provoca que se releguen a los últimos puestos del ranking general. La aplicación debe mostrar los dorsales de los corredores que han conseguido las medallas de oro, plata y bronce.
- 5.20.** La fusión de dos tablas ordenadas consiste en copiar todos sus elementos (de ambas tablas) en una tercera que deberá seguir ordenada. Podemos realizar una fusión «ineficiente» copiando los elementos de ambas tablas (sin tener en cuenta el orden) en la tabla final y ordenar esta. Existe una manera óptima de realizar la fusión en la que se elige en cada momento el primer elemento no copiado de alguna de las tablas y se añade a la tabla final, que seguirá ordenada sin necesidad de ordenación alguna.

Busca información sobre el algoritmo de fusión e impleméntalo en Java.

## Actividades de ampliación

- 5.21.** Explica las ventajas e inconvenientes de utilizar tablas. Busca algunas alternativas al uso de tablas para almacenar datos.
- 5.22.** Busca en internet información sobre el algoritmo de ordenación de la burbuja y explica su funcionamiento.
- 5.23.** El algoritmo de ordenación por selección ordena una tabla seleccionando en cada momento el mayor y menor elemento de entre los que no se encuentran ordenados. Busca información sobre este algoritmo e impleméntalo.
- 5.24.** Halla información sobre el algoritmo de ordenación por intercambio, que es muy parecido al algoritmo de la burbuja. Tras indagar sobre él, explica las similitudes y diferencias entre ambos tipos de algoritmo.
- 5.25.** Con respecto a los algoritmos de ordenación por intercambio y el de la burbuja, para una tabla que está prácticamente ordenada, ¿cuál de los dos sería mejor utilizar? Razona tu respuesta.
- 5.26.** Las pilas son estructuras de datos que simulan una pila de platos, donde los datos entran y salen por la parte superior. Se pueden implementar de muchas formas, pero es habitual hacerlo mediante una tabla. Busca información sobre ellas e implementa sus operaciones utilizando una tabla.
- 5.27.** Al igual que las pilas, las colas son una estructura de datos que simula una cola de espera, donde hay una persona esperando el primero y el resto aguarda su turno. Busca información sobre esta estructura de datos e implementa sus operaciones utilizando una tabla.
- 5.28.** Por simplicidad, solo hemos visto las tablas bidimensionales cuyas columnas contienen el mismo número de elementos. Es posible crear tablas irregulares, donde cada columna tiene un número determinado de elementos. Busca información de cómo se crean este tipo de matrices y abrid un debate entre toda la clase donde cada uno aporte el contenido de interés que ha encontrado.