UD04.1.Estructuras de Almacenamento. Arrays Apuntes

DAM1-Programación 2024-25

Otras fuentes:

- **Programación Java: Teoría**
  - Arrays unidimensionales en Java
  - Matrices en Java (arrays irregulares)
  - Copiar arrays en Java (`clone()`)
  - Arrays y métodos en Java
  - Algoritmos de ordenación. Metodo de la Burbuja
- **Operaciones con ArraysBidimensionales.java**

Introducción

Arrays/Tablas/Vectores/Arreglos.
Índices. Construcción de tablas. Longitud y tipo.
Operador new. Referencias (null). Recolector de basura.

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`.

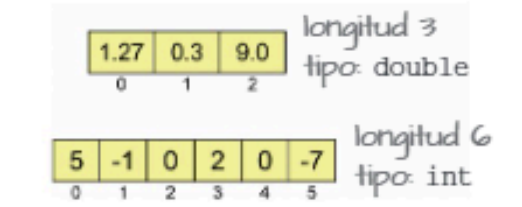


Figura 5.3. *Tablas con distintas longitudes y tipos.*

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.

1. Crear tablas de objetos, por ejemplo de clase Persona.
2. Investiga sobre “tablas dinámicas” en Java o ArrayList que veremos en Unidades posteriores:
    a. Java ArrayList. Estructura dinámica de datos
    b. ArrayList de Objetos en Java

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`. Estudiemos 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`:

   tamaño en memoria = *n.*° elementos tabla × tamaño tipo (int) = 10 × 4 bytes = = 40 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.

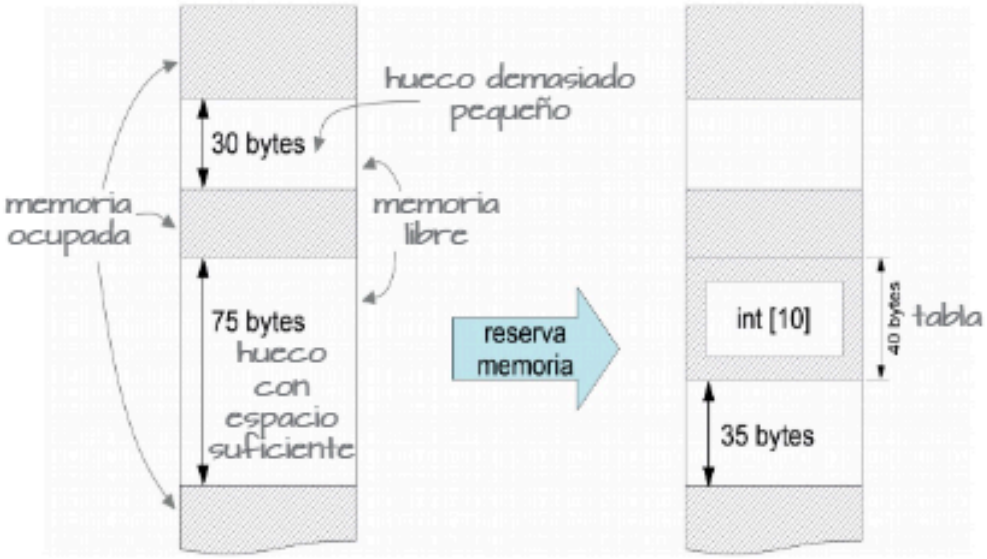


Figura 5.4. *Reserva de memoria. Representación de la memoria antes y después de construir una tabla de 10 enteros.*

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.

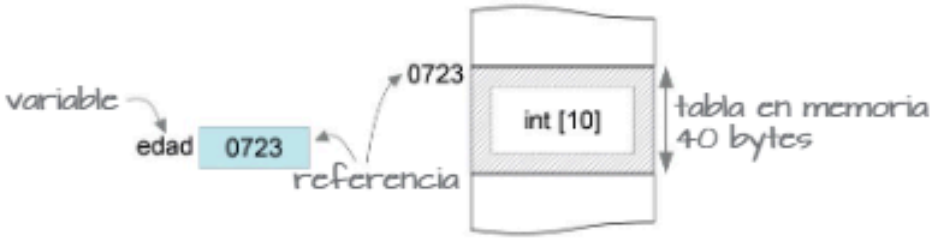


Figura 5.5. *Asignación de una referencia a una variable. La variable* `edad` *contiene la referencia que le permite acceder a una posición de memoria y encontrar ahí la tabla con todos sus datos.*

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.*

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

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.

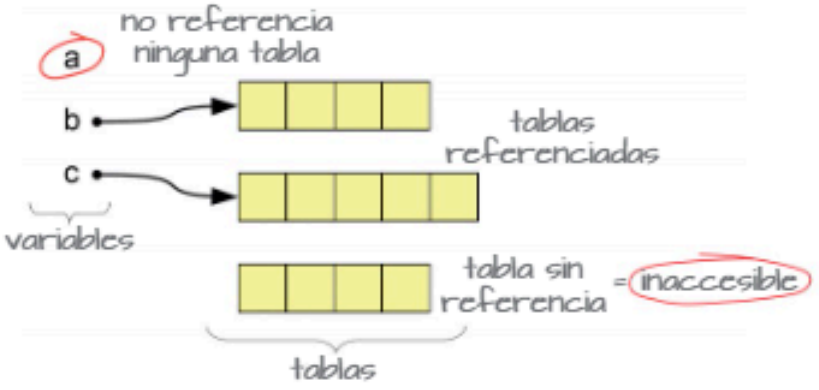


Figura 5.7. *La tabla que no está referenciada tan solo ocupa espacio en la memoria y es completamente imposible acceder a sus datos. Java se encarga de liberar la memoria ocupada de forma inútil.*

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
```

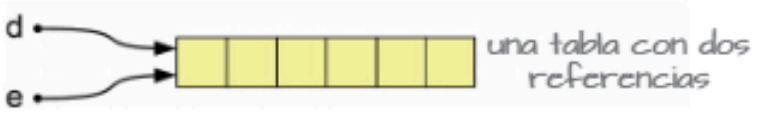


Figura 5.8. *Tabla multirreferenciada. En programación es muy habitual utilizar más de una variable que referencien los mismos elementos. Este mecanismo es la base para compartir información entre distintas partes de un programa.*

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
```

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

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 referencie— 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
```

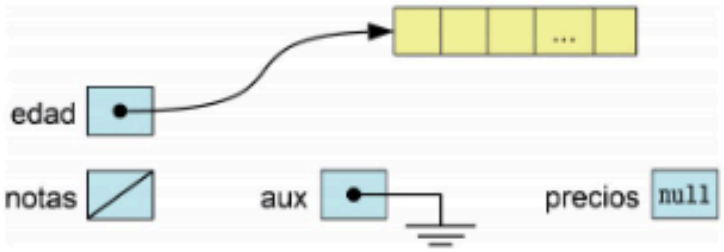


Figura 5.9. *Mientras una referencia se suele representar con una flecha entre la variable y la tabla, es habitual encontrar la referencia vacía representada mediante una línea cruzada, una toma de tierra o incluso con la propia palabra* `null`. *Todo ello informa que la variable no está referenciando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

Uso de tablas

Tablas ordenadas. Tablas como parámetros. Clase Arrays. Longitud, recorrido, ordenación, búsqueda, copia

En Java, los arrays o tablas mantienen su **longitud constante y no es posible cambiar el número de elementos** que contienen. Por eso, si necesitamos modificar la longitud de una tabla, lo que haremos será crear una segunda tabla con el número de elementos necesarios y crear en ella los datos que nos interesan de la tabla original.

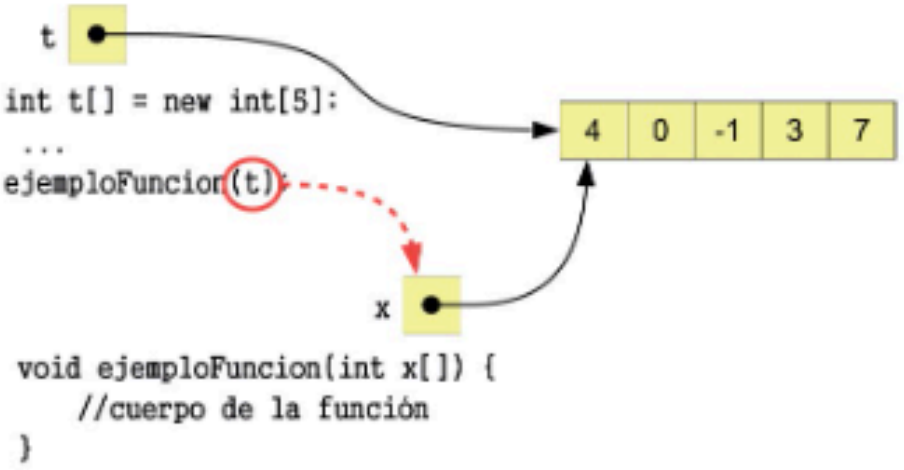
Tablas como parámetros de funciones

- Arrays unidimensionales en Java
- Arrays y métodos en Java

Al invocar una función con parámetros el valor de la variable que se utiliza en la llamada (parámetro actual) se copia al parámetro (formal) de la función, que funciona como una variable local en la función.

Cuando utilizamos tablas como parámetros el mecanismo es el mismo pero, en este caso, se copia la referencia de la tabla, con lo que la tabla estará referenciada también por el parámetro local. Así, al modificar un elemento de la tabla dentro de la función, la modificación se mantendrá al terminar la función.

En el ejemplo de la siguiente figura, si dentro de la función `ejemploFuncion()` ejecutamos `x[2] = 10`; estamos cambiando también `t[2]`, ya que tanto `x` como `t` referencian la misma tabla.



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

ArrayNumeros. Introduce un número **n** por teclado. A continuación, solicita al usuario que teclee **n** números y almacénalos en un array. A continuación realiza y muestra la media de los números positivos, la media de los negativos y cuenta el número de ceros introducidos.

GeneraArrayAleatorio. Escribe una función que genere y devuelva un array de n números aleatorios entre -9 y 9, incluyendo el 0.

```
static int[] arrayAleatorio(int n)
```

FuncionContar. Escribe una función que cuente las veces que aparece un valor clave en una tabla de enteros.

```
static int contar(int t[], int clave)
```

FuncionBuscar. Escribe una función que busque 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.

```
int buscar(int t[], int clave)
```

Nota: Puedes ir recopilando todas estos métodos estáticos en una clase de utilidades, por ejemplo, `ArraysUtil`, para invocarlas cómodamente desde otros programas cuando las necesites.

Operaciones con tablas: la clase Arrays

- Clase Arrays en la API de Java

La API de Java proporciona 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;
```

Podemos implementar nuestras propias operaciones para tablas, pero Arrays nos da la seguridad de un código eficiente, sin errores y la comodidad de ahorramos tiempo. Siempre que sea posible aprovecharemos las funcionalidades presentes en la clase Arrays.

Número de elementos de una tabla

Los arrays en Java disponen del atributo **length** para conocer el número de elementos, o longitud, con el que se construyó.

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
```

Una situación habitual en la que necesitamos averiguar la longitud de un array es cuando lo pasamos como parámetro a una función, ya que en ese contexto desconocemos la longitud con la que se creó.

Inicialización

Podemos inicializar los elementos de un array a un valor dado utilizando el método **fill()** de la clase **Arrays**.

- `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.

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**, sin incluír éste.

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
}
```

Sintaxis bucles for-each

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;
}
```

E0501. Crea una tabla de longitud 10 que se inicializará con números aleatorios comprendidos entre 1 y 100. Muestra la suma de todos los números aleatorios que se guardan en la tabla.

Genera dos métodos para resolver el problema.

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()`.

E0502. Diseña un programa que solicite al usuario que introduzca por teclado 5 números decimales. A continuación muestra los números en el mismo orden en que se han introducido.

E0503. Escribir un programa que solicite al usuario cuántos números desea introducir. A continuación introducir por teclado esa cantidad de números enteros. Por último, mostrar los números en el orden inverso al introducido.

E0504. Diseñar la función: `static int maximo(int t[])` que devuelva el máximo valor contenido en la tabla `t`.

Ordenación

- Como ordenar arrays en Java. Método Arrays.sort()
  - Collections.reverseOrder()
- Algoritmos de ordenación.
  - Metodo de la Burbuja
  - Ordenación por Selección
  - Ordenamiento por Inserción directa
  - Quicksort Entre O(n2) y O(n log n)
  - MergeSort Promedio O(n log n)

- Rendimiento de algoritmos y notación Big-O | campusMVP.es

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.

E0505. Escribe la función: `static int[] rellenaPares(int longitud, int fin)` que crea y devuelve una tabla ordenada de la longitud especificada rellena con números pares aleatorios en el rango desde 2 hasta el valor de `fin` inclusive.

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 clave secuencialmente en una tabla no ordenada. Devuelve la posición de la primera aparición de la clave o el valor -1 si no se encuentra.

```
static int buscar(int t[], int clave) {
    for(int i = 0; i < t.length; i++)
        if (t[i] == clave)
            return i;
    return -1;
}
```

BuscarUltimo. Realiza una versión del método anterior que busque en el array la última aparición de la clave.

BuscarDesde. Realiza otra versión que busque la clave a partir de una posición concreta del array.

Búsqueda en una tabla ordenada
En este caso, aprovechamos que la tabla está ordenada para hacer una búsqueda más eficiente.

Esta propiedad de las tablas ordenadas se aprovecha en el algoritmo de búsqueda binaria, o dicotómica, 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.

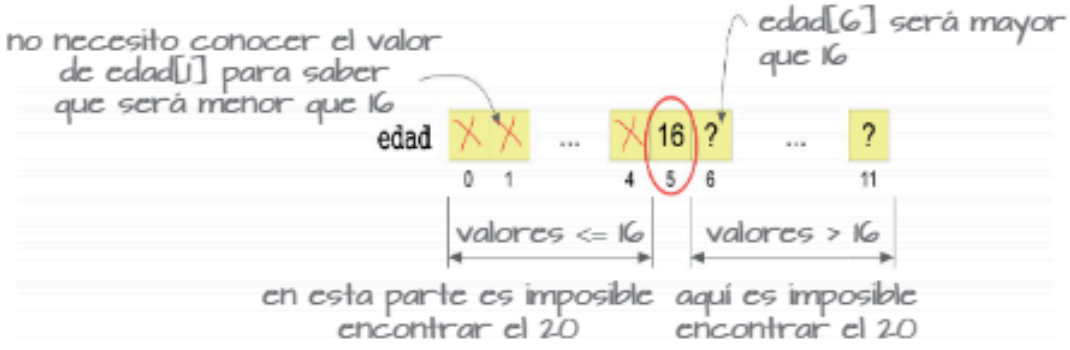


Figura 5.12. *Búsqueda dicotómica. El elemento central de una tabla proporciona información extra de dónde buscar.*

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

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

Cuando el elemento a 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 indice de inserción se calcula con la siguiente fórmula:

```
int indiceInsercion = -posicion - 1;
```

siendo `pos` el valor negativo devuelto por `binarySearch()`.

E0506_Primitiva. 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.

```
static int numAciertos(int[] apuesta, int[] ganadora)
```

Crea un método que devuelve una tabla de números enteros aleatorios entre dos números y de una longitud especificada.

```
static int[] tablaAleatoria(int numInicio, int numFin, int longitud);
```

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 claveBusqueda)`: 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`.

Copia

- Copiar arrays en Java (`clone()`)

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.

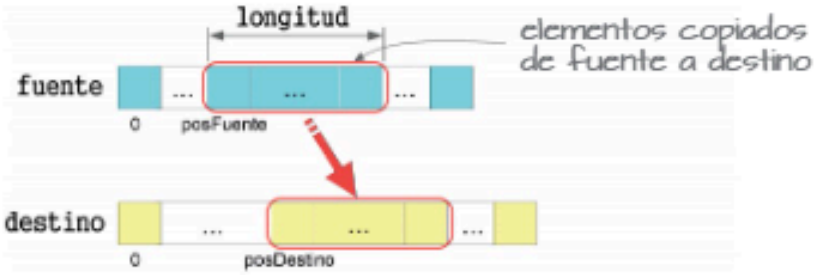


Figura 5.13. *Proceso de copia que realiza* `copyarray()`. *Los elementos copiados pueden moverse desde su posición en la tabla fuente (u origen) a cualquier otra posición en la tabla destino. Ambas tablas pueden ser de distinta longitud, aunque siempre hay que estar seguro de que no copiaremos en elementos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

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;
```

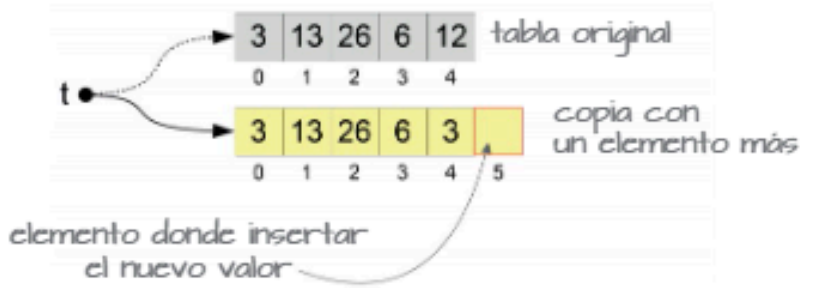


Figura 5.14. *Inicialmente la tabla original estaba referenciada por* `t`. *Tras ejecutar* `Arrays.copyOf()`, `t` *se modifica y pasa a referenciar a la nueva tabla, que es una copia con una longitud incrementada en uno. Ahora disponemos de un nuevo elemento para añadir un dato más.*

E0507. 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.

Implementa una función que genere y devuelva un array de n números enteros aleatorios usando el siguiente prototipo:

```
int[] tablaRandom(int n)
```

E0508. Genera una tabla de n números enteros aleatorios y construye a partir de ella otras dos tablas, una con los valores pares y otra con los valores impares. Muestra las tablas de pares e impares ordenadas.
Piensa dos versiones:

1. Sin modificar la tabla original
2. Pudiendo modificar la tabla original

Implementa sendas funciones para los procesos anteriores usando los siguientes prototipos:

```
int[] getPares(int t[])
int[] getImpares(int t[])
```

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 (indiceInsersion < 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, indieInsercion+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
```

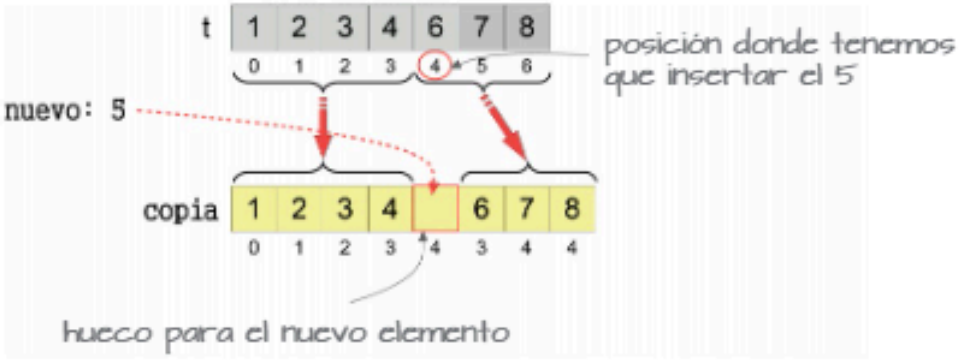


Figura 5.15. *Momento en el que hemos creado el hueco para el nuevo elemento y hemos copiado los datos. Solo queda insertar el nuevo elemento (5) en el hueco y referenciar la tabla* `copia` *con* `t`*, dando la sensación de que la tabla ha incrementado su longitud. Es importante que, tras todas las operaciones,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

Escribe una función para insertar en un array ordenado de números enteros:

```
static public int[] insertarOrdenado(int[] t, int num);
```

E0509. Diseña una aplicación para gestionar un campeonato de programación donde se introduce la puntuación obtenida por 5 programadores, conforme van terminando la prueba, en forma de números enteros. 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 cuyas puntuaciones se incluyen con el resto. La forma de especificar que no intervienen más programadores de exhibición es introducir como puntuación un -1. Por último, la aplicación debe mostrar los puntos ordenados de todos los participantes.

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
}
```

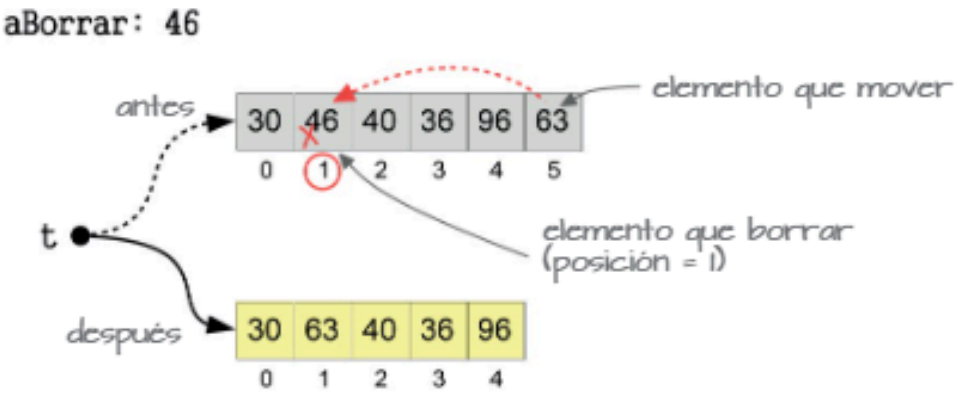


Figura 5.16. *Todas las operaciones de eliminación comienzan localizando el elemento que se va a borrar. Después se han de recolocar el resto de elementos para que produzcan el efecto de que el elemento en cuestión ha desaparecido. Finalmente, redimensionamos a una tabla más pequeña.*

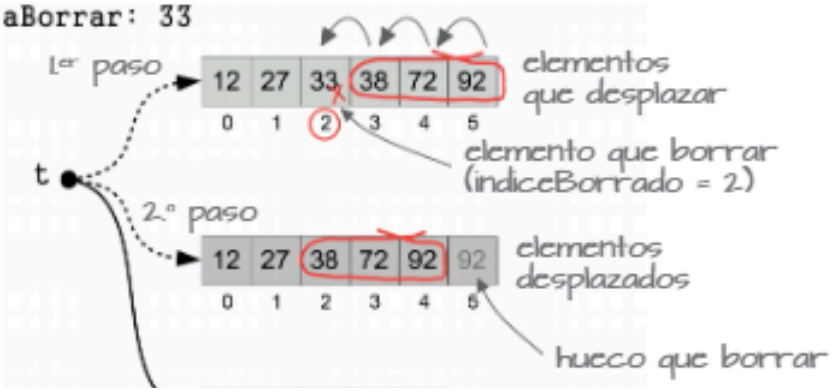
E0510. Escribir la función:

```
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`.

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.



Escribe una función para eliminar un valor en un array ordenado de números enteros:

```
static public int[] borrarOrdenado(int[] t, int num);
```

Crea una variante del método anterior que borre todas las ocurrencias de un valor en caso de que esté repetido.

```
static public int[] borrarTodosOrdenado(int[] t, int num);
```

Comparación de 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
```

E0511. Desarrollar el **juego "la cámara secreta"**, que consiste en abrir una cámara mediante su combinación secreta, que está formada por una combinación de dígitos del 1 al 5. Al inicio el jugador especificará la longitud de la combinación, de modo que 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 que se encuentra en el mismo puesto en la combinación secreta.

Arrays paralelos: Investiga qué son arrays paralelos y para qué sirven con el siguiente ejemplo/tutorial.

A partir del código del ejemplo, crea dos arrays paralelos para almacenar los nombres y las edades asociadas en el ejemplo, muéstralos en pantalla.
Implementa una función que ordene los dos arrays por un criterio (por ejemplo, edad ascendente).

ValidarDNI: Crea una función con el siguiente prototipo que utilice arrays para validar que un DNI y su letra son correctos:

```
boolean esValidoDNI(int numero, char letra)
```

CÓDIGO PARA LA LETRA DEL D.N.I. O DEL N.I.F.																					
RESTO	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
LETRA	T	R	W	A	G	M	Y	F	P	D	X	B	N	J	Z	S	Q	V	H	L	C	K	E

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.

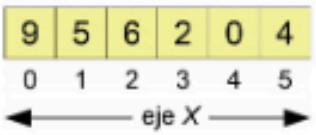


Figura. 5.18. *El eje X, o eje de abscisa, es el eje horizontal. Hay que entender que esto es una representación, y lo habitual es que nos figuremos que las tablas unidimensionales se expanden horizontalmente.*

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.

Arrays bidimensionales o Matrices

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 × 5) 25 elementos.

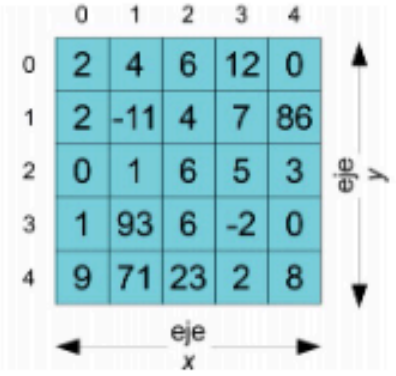


Figura 5.19. *Matriz de 5 x 5 elementos. Ahora la identificación de cada elemento viene dada por el índice del eje X y el índice del eje Y.*

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));
```

E0512. Crea una tabla bidimensional de longitud 5 x 5 y rellénala de la siguiente forma: el elemento de la posición [n][m] debe contener el valor 10 x n + m. Después se debe mostrar su contenido.

- Matrices en Java

Comparar Matrices (Arrays.deepEquals():

1. Comparar Matrices en Java - Línea de Código

Arrays de 3 dimensiones

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.

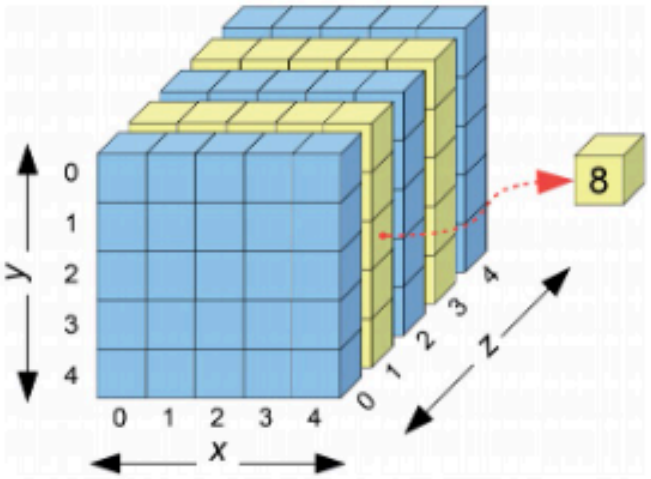


Figura 5.20. *Una tabla tridimensional se representa mediante un cubo. Se aprecia cómo se necesitan tres dimensiones (tres índices) para poder identificar cualquier elemento, que contiene un dato del tipo del que está declarada la tabla. En nuestro ejemplo, el índice [4][2][1] (que corresponde a: x,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

Arrays de más de 3 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	4	5
	<100 g	100-200 g	200-300 g	300-400 g	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;
```

Resumen

Declarar, iniciar, recorrer un array

```
// Declaración de un array
tipo nombreArray[];

// Creación de un array
nombreArray = new tipo[longitudArray];

// Declaración y creación (en una línea)
tipo nombreArray[] = new tipo[longitudArray];

// Asignación de valores a elementos de un array
nombreArray[indice] = valor;

// Declaración, creación e inicialización de un array de n
elementos (en una línea)
tipo nombreArray[] = {valorElem0, valorElem1, …, valorElemN-1}

// Longitud de un array
nombreArray.length

// Recorrer un array (con un bucle for)
for (int i = 0; i < nombreArray.length; i++) {
    // Operación de lectura o escritura sobre cada
    // elemento del array: nombreArray[i]
}

// Recorrer un array (con bucle un foreach)
for (tipo variable : nombreArray) {
    // Accedemos a cada elemento del array con la variable
}

// Clase Arrays: métodos
fill()
toString()
deepToString()
sort()
binarySearch()
equals()
deepEquals()
copyOf()
copyOfRange()
...
System.arraycopy()

etc.
```