CFGS APLICACIONES MULTIPLATAFORMA - PROGRAMACIÓN

UNIDAD 4

ESTRUCTURAS DE ALMACENAMIENTO

DIEGO VALERO ARÉVALO
BASADO EN APUNTES Y EJERCICIOS PROPORCIONADOS POR
Mª CARMEN DÍAZ GONZÁLEZ - IES VIRGEN DE LA PALOMA

U4 - **ESTRUCTURAS DE ALMACENAMIENTO**
ÍNDICE

4.1 - INTRODUCCIÓN A ARRAYS

Los **arrays** o **tablas** son **estructuras de almacenamiento** que podríamos resumir como **"variables que contienen múltiples valores"**.

Son un tipo de colección en el cual **se almacenan datos del mismo tipo** que se colocan en **posiciones numeradas**. Son útiles para **agrupar** variables en vez de declararlas todas por separado.

Vamos a ver dos tipos de arrays:

Unidimensionales	Multidimensionales o matrices

>> 4.1.1- FUNCIONAMIENTO BÁSICO DE UN ARRAY

Imagina que tenemos una **mesa** con varios **cajones** en fila que están numerados y empiezan por el **número 0**, y que cada uno de esos cajones sólo puede tener una cosa.

Pero no podemos empezar a meter cosas en los cajones **hasta que no especifiquemos de qué tipo son estas**, para poder saber exactamente qué es lo que vamos a guardar. Por ejemplo, si queremos guardar una manzana, una naranja, una pera,... tendremos que saber que la mesa será sólo para frutas.

4.2 - ARRAYS UNIDIMENSIONALES

>> 4.2.1- DECLARAR UN ARRAY UNIDIMENSIONAL

Si trasladamos el ejemplo de la mesa a programación, lo que queremos hacer es un **array**, el cual se empieza a construir de la siguiente manera:

```
tipo[] nombreArray; / tipo nombreArray[];
```

Para especificar que una variable es un array, se usa `[]`, que se puede poner **detrás del tipo (recomendada)** o **del nombre**, es indiferente.

Ahora, los arrays tienen un **tamaño predefinido** para saber cuántas cosas pueden guardar. Podemos establecerlo de dos maneras:

POR CANTIDAD DE DATOS	POR TAMAÑO n
`tipo[] nombreArray = {a, ..., n};`	`tipo[] nombreArray = new tipo[n];`

Dependiendo de cuántos datos por defecto añadamos al array, este automáticamente registrará esa cantidad como el tamaño máximo.	Asignando esta estructura donde `n` es un número definimos el tamaño directamente.

Ahora que ya sabemos cómo funciona un array, podemos transformar la mesa a un diagrama y crear un ejemplo:

EJEMPLO

"manzana"	"naranja"	"pera"	"kiwi"	"plátano"	"sandía"
0	1	2	3	4	5

El tamaño de la mesa es **6**, y queremos guardar una **fruta** en cada cajón, las cuales queremos que sean de tipo **string**.

Podemos declararlo con el **tamaño** y añadir las frutas después o **directamente** con las frutas que queramos:

.java

```java
String[] frutas = new String[6];
```

```java
String[] frutas = {"manzana","naranja","pera","kiwi","plátano","sandía"};
```

>> 4.2.2- AÑADIR VALORES A UN ARRAY UNIDIMENSIONAL

Si sabemos el tamaño de nuestro array y el tipo de datos, pero aún **no sabemos qué vamos a guardar**, declararemos entonces el array por su tamaño, pero ¿cómo guardamos las cosas después? Tendremos que inicializar el array, y para ello hay que acceder a sus **posiciones**.

```java
nombreArray[x] = valor;
```

`nombreArray[]`	Para acceder a una posición del array, escribimos su nombre junto a `[]`.
`x`	Dentro de `[]` escribimos el **número de la posición (`x`)** donde queremos guardar un `valor`. ⚠ El número de la posición **nunca puede ser mayor o igual al tamaño del array**, porque no existe en el array y saltaría una excepción `OutOfBounds` **(fuera de los límites)**.

EJEMPLO

Persona.java

```java
frutas[1] = "naranja";
```

Aquí hemos añadido al **segundo elemento** (cuyo **número de posición** es `1`) del array **frutas**, el valor "naranja".

💡 La **última posición** de un array siempre será **su tamaño - 1**. Para **saber el tamaño de un array**, este trae consigo un método que devuelve ese número: `.length`.

EJEMPLO

.java	Console

```java
1  System.out.println(frutas.length);
```
```
6
```

>> 4.2.3- ACCEDER A LOS ELEMENTOS DE UN ARRAY UNIDIMENSIONAL

Para acceder a un elemento hacemos exactamente igual que cuando queremos añadir un valor: llamamos al **array** y al **número de la posición** que nos interesa.

```java
nombreArray[x];
```

EJEMPLO

.java
```java
1  System.out.println(frutas[1]);
```
Console
```
naranja
```

>> 4.2.4- RECORRER UN ARRAY UNIDIMENSIONAL

Es muy común que necesitemos recorrer todos los elementos de un array para, por ejemplo, comprobar si un dato existe dentro del array o simplemente si queremos imprimir por consola cada elemento. Para ello podemos usar un bucle **`for`** que comience en la primera posición (`0`) y llegue hasta **antes de alcanzar el tamaño** o hasta **su tamaño-1**:

```java
for(int i = 0; i < nombreArray.length; / i <= nombreArray.length-1; i++){
  ...
}
```

EJEMPLO

.java
```java
1  for(int i = 0; i < frutas.length; i++){
2     System.out.println(frutas[i]+", ");
3  }
```
Console
```
manzana, naranja, pera, kiwi,
plátano, sandía,
```

>> 4.2.5- BÚSQUEDAS Y ORDENACIONES DE UN ARRAY UNIDIMENSIONAL

Buscar en un array tiene mucho que ver con **ordenarlo**, ya que en un array no ordenado solo hay una manera de buscar un elemento: recorrer el array comprobando uno a uno.

JAVA tiene predefinidos en la clase **`Arrays`** varios métodos que ordenan estos automáticamente.

TEN EN CUENTA

Las búsquedas y ordenaciones son operaciones que necesitan muchos recursos, y cuanto más grande sea el array que vamos a utilizar, más tiempo y memoria requerirá. **Estas operaciones no son recomendadas para arrays grandes.**

· 4.2.5.1- BÚSQUEDAS

Tenemos **dos métodos de búsqueda** principales:

Secuencial	Dicotómica

BÚSQUEDA SECUENCIAL

La **búsqueda secuencial** es la más fácil de las dos ya que consiste en **comparar los elementos del vector (`nombreArray[i]`) con el elemento a buscar**.

```
for(int i = 0; i < nombreArray.length; i++){
   if(nombreArray[i] == elemento){
      ...
   }
}
```

Un ejemplo de uso de este método es **almacenar la posición del vector** cuando este coincida con el elemento.

EJEMPLO

.java

```
1  int[] arrayNumeros = {4,3,2,5,1};
2  int elemento = 5;
3  int posicion;
4
5  for(int i = 0; i < arrayNumeros.length; i++){
6     if(arrayNumeros[i] == elemento){
7        posicion = arrayNumeros[i];
8     }
9  }
10
11 System.out.println("La posición del elemento es "+posicion);
```

Console

```
La posición del elemento es 3
```

BÚSQUEDA DICOTÓMICA

La **búsqueda dicotómica** divide un array en dos y lo recorre **desde la izquierda** o **desde la derecha hacia el centro** hasta **encontrar** el elemento. Esto se define calculando el valor más cercano al centro y comparándolo con el elemento.

```
int[] array = ...;
int elemento = n;
int izda = 0;
int dcha = array.length-1;
int centro = (izda+dcha)/2;
int posicion = -1;

while (izda<=dcha && array[centro] != elemento){
   if(n<array[cent]) {
      dcha = centro-1;
   }
   else {
      izda = centro+1;
   }
   cent = (izda+dcha)/2;
}

if(izda>dcha) {
   posicion = -1;
}
else {
   posicion = centro;
}
```

La única **restricción** para usar la búsqueda dicotómica es **que el array tiene que estar ya ordenado**.

Un ejemplo de uso de este método es **almacenar la posición del vector** cuando este coincida con el

elemento.

EJEMPLO

.java

```java
int[] arrayNumeros = {1,2,3,4,5,6,7,8,9,10};
int elemento = 2;
int izda = 0;
int dcha = array.length-1;
int centro = (izda+dcha)/2;
int posicion = -1;

while (izda<=dcha && array[centro] != elemento){
   if(n<array[cent]) {
      dcha = centro-1;
   }
   else {
      izda = centro+1;
   }
   cent = (izda+dcha)/2;
}

if(izda>dcha) {
   posicion = -1;
}
else {
   posicion = centro;
}

if(posicion = = -1) {
   System.out.println("El elemento no está en el array");
}
else {
   System.out.println("La posición del elemento es "+posicion);
}
```

Console

```
La posición del
elemento es 1
```

· 4.2.5.2- ORDENACIONES

Existen **cuatro métodos de ordenación** principales:

Bubble	Insertion	Selection	Quicksort

Aunque son interesantes de conocer, no es necesario saber cómo funcionan a fondo ya que JAVA los trae implementados por defecto. Las ordenaciones de **`arrays`** pueden hacerse con el método **`.sort`** que trae la propia clase **`Arrays`**.

```java
Arrays.sort(nombreArray);
```

EJEMPLO

.java

```java
int[] arrayNumeros = {8,7,9,6,4,5,2,3,1,10};

for(int i = 0; i < arrayNumeros.length; i++){
   System.out.println("Antes de ordenar:");
   System.out.println(arrayNumeros[i]+", ");
}

Arrays.sort(arrayNumeros);
System.out.println();

for(int i = 0; i < arrayNumeros.length; i++){
   System.out.println("Después de ordenar:");
   System.out.println(arrayNumeros[i]+", ");
}
```

Console

```
Antes de ordenar:
8, 7, 9, 6, 4, 5, 2, 3, 1, 10,

Después de ordenar:
1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
```

Vamos a ver la clase **`Arrays`** más a fondo el siguiente apartado.

>> 4.2.6- MÉTODOS DE LA CLASE `ARRAYS`

Vamos a ver algunos de los **métodos más utilizados** dentro de esta clase que nos permite manipular **`arrays`** más fácilmente:

`.fill(array, valor);`	Rellena un **`array`** con el mismo **`valor`** en todas las posiciones.
`.equals(arrayA, arrayB);`	Compara **`arrayA`** con **`arrayB`** y devuelve un **booleano**. Se consideran iguales si son del mismo tipo, tamaño y contienen los mismos valores, da igual cómo estén colocados.

`.sort(array);`
 	
`.sort(array, posA, posB);`
	Como hemos visto en el apartado anterior, ordena los elementos de un **`array`** en orden ascendente utilizando el alfabeto ASCII.
	
	También podemos ordenar **sólo una parte del array**, especificando las posiciones desde la **`posA`** hasta la **`posB`**.

`.binarySearch(array, elem);`	Permite buscar un **`elemento`** de forma ultrarrápida en un **`array`** **ordenado**.

4.3 - ARRAYS MULTIDIMENSIONALES O MATRICES

>> 4.2.1- DECLARAR UN ARRAY MULTIDIMENSIONAL O MATRIZ

Podemos almacenar **arrays dentro de arrays**, esto es lo que se conoce como **arrays multidimensionales o matrices**. Para crear uno lo definimos de la siguiente manera:

```
tipo[][] nombreMatriz ; / tipo nombreMatriz [][];
```

Al igual que con los array, se usa **`[][]`**, que se puede poner **detrás del tipo (recomendada)** o **del nombre**, es indiferente.

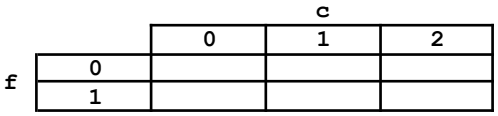
No podemos rellenar una matriz directamente, así que lo siguiente es definir el tamaño de las **filas** y de las **columnas**:

```
tipo[][] nombreMatriz = new tipo[f][c];
```

EJEMPLO

.java

```
1  int[][] miMatriz = new int[2][3];
```



>> 4.2.2- AÑADIR VALORES A UNA MATRIZ

Para acceder a las posiciones de una matriz debemos **especificar** el número de la **fila** y la **columna**.

```
nombreMatriz[f][c] = valor;
```

`nombreMatriz[][]`	Para acceder a una posición del array, escribimos su nombre junto a `[][]`.
`[f][c]`	Dentro de `[][]` escribimos el **número de la fila (`f`)** y **de la columna (`c`)** donde queremos guardar un **`valor`**.

> ⚠ El número de las posiciones **nunca puede ser mayor o igual al tamaño de las filas** o **de las columnas** respectivamente, porque no existen en la matriz y saltaría una excepción **`OutOfBounds` (fuera de los límites)**.

EJEMPLO

.java

```
1  miMatriz[0][1] = 7.5;
2  miMatriz[1][1] = 12;
3  miMatriz[2][0] = 3.63;
```

f \ c	0	1	2
0		7.5	
1	3.63	12	

>> 4.2.3- ACCEDER A LOS ELEMENTOS DE UNA MATRIZ

Para acceder a un elemento hacemos exactamente igual que cuando queremos añadir un valor: llamamos a la **matriz** y al **número de la fila y de la columna** que nos interesa.

```
nombreMatriz[f][c];
```

EJEMPLO

.java

```
1  System.out.println(miMatriz[0][2]);
```

Console

```
null
```

>> 4.2.4- RECORRER UNA MATRIZ

La manera de recorrer una matriz es a través de **dos bucles `for` anidados**: el primero recorrerá las **filas** y el segundo las **columnas de esa fila**.

```
for(int fila = 0; fila < nombreMatriz.length; fila++){
   for(int columna = 0; columna < nombreMatriz[fila].length; columna++){
      ...
   }
}
```

El **`.length`** por defecto de una matriz devuelve el **número de filas**, y si accedemos a **cualquiera de las filas** podremos saber **el número de columnas totales**, porque todas las filas son igual de largas.

Esto es muy útil para rellenar las posiciones de una matriz con valores.

4.4 - CLASE STRING

La clase **`String`** nos permite trabajar con conjuntos de caracteres complejos. Ya sabes que en esencia un **`String`** es un **`array`** de **`char`**.

Puedes **concatenar** varias **líneas de `strings`** o **`strings` y otras variables** con el operador **+**.

EJEMPLO

.java

```
1  String texto = "Este es un texto que ocupa " +
2                 "varias líneas, pero lo podemos " +
3                 "concatenar perfectamente";
```

>> 4.4.1- COMPARACIÓN (MÉTODOS `EQUALS` Y `COMPARE TO`)

Como los objetos String son diferentes a otros tipos de variables, **no podemos usar el operador == para compararlos**. En su lugar tenemos los siguientes métodos:

`stringA.equals(stringB);`	El resultado es true si **`stringA`** es igual a **`stringB`**. Ambas son variables de tipo **`String`**.
`stringA.equalsIgnoreCase(stringB);`	Igual que la anterior, pero ignora las mayúsculas y minúsculas.
`stringA.compareTo(stringB);`	Compara ambas cadenas, considerando el **orden alfabético ASCII** (donde la letra **ñ y las tildes** son mayores que otras letras). Si **`stringA`** es **mayor en orden alfabético** que **`stringB`**, devuelve **un número mayor que** `0`, si son **iguales devuelve** `0` y si **es la segunda la mayor, devuelve un número menor que** `0`.
`stringA.compareToIgnoreCase(stringB);`	Igual que la anterior, pero ignora las mayúsculas y minúsculas.

>> 4.4.2- MÉTODOS DE LA CLASE `STRING`

Vamos a ver algunos de los **métodos más utilizados** dentro de esta clase que nos permite manipular **`strings`** más fácilmente:

`.valueOf(var);`	Convierte el **tipo** de **`var`** a **`String`**.
`string.length(string);`	Devuelve la longitud del **`string`**.
`stringA + stringB;` `stringA.concat(stringB);`	Para **concatenar** cadenas. Disponemos del operador **+** o del método **`.concat`**.
`string.charAt(n);`	Devuelve el **`char`** que se encuentra en la **posición** **`n`** del **`string`**.
`string.substring(posA, posB);`	Devuelve la porción de **`string`** desde **`posA`** **hasta la anterior a** **`posB`**. Si las posiciones **no existen** en el **`string`** saltaría una excepción **`OutOfBounds`** **(fuera de los límites)**.

`stringA.indexOf(stringB);`
 	
`stringA.indexOf(stringB, pos);`
	Devuelve la **primera posición** en la que aparece **`stringB`** en **`stringA`**. En el caso de que **`stringB`** no se encuentre, devolverá **`-1`**. **`stringB`** puede ser **`char`** o **`String`**.
	
	También podemos especificar a partir de qué **posicion** (**`pos`**) queremos que busque.

`stringA.lastIndexOf(stringB);`
 	
`stringA.lastIndexOf(stringB, pos);`
	Devuelve la **última posición** en la que aparece **`stringB`** en **`stringA`**, ignorando los anteriores si se repiten.
	
	También podemos especificar a partir de qué **posicion** (**`pos`**) queremos que busque.

`stringA.endsWith(stringB);`
 	
`stringA.startsWith(stringB);`
	Estos métodos devolverán **true** si **`stringA`** termina (**ends**) o empieza (**starts**) con **`stringB`** .

`string.replace(stringA, stringB);`
 	
`string.replaceFirst(stringA, stringB);`
 	
`string.replaceAll(stringA, regex);`
 	
Estos métodos son muy útiles para quitar algún carácter en específico. La sustitución puede incluso hacerse con un string vacío (`""`).
	Cambia **todas las apariciones de un `char` o `String` en `stringA`** por **lo que se especifique en `stringB`** y lo almacena como resultado. **El texto original no se cambia**, por lo que hay que asignar el resultado a un nuevo **`String`** para conservar el texto cambiado.
	
	`.replaceFirst` sólo afecta a la primera coincidencia, y **`.replaceAll`** se utiliza si en vez de otro **`string`** queremos usar **expresiones regulares**.

`string.toUpperCase();`
 	
`string.toLowerCase();`
	Convierte **todos los caracteres de `string`** a **mayúsculas** (**`Upper`**) o a **minúsculas** (**`Lower`**).

`string.toCharArray();`	Convierte un **`string`** a un **`array`** de tipo **`char`**. Esto nos permite **manipular un `string` de forma más fácil** usando los métodos de la clase **`Arrays`**.
`.format("%.nf", var);`	Modifica el **formato del `string` a mostrar**. Muy útil para mostrar **sólo los decimales que necesitemos de un número decimal**. Indicaremos **`%`** para indicar **la parte entera** más **el número de decimales a mostrar (`n`) seguido de `f`**.
`string.split(regex);`	Devuelve un **`array`** de tipo **`String`**, el cual almacenará en cada posición un **fragmento del `string`**, que se divide en cada parte que coincida con ***`regex`***, que puede ser una expresión regular, una letra, un signo,... Esta parte **no aparece en el `array` final**.
`string.trim();`	Devuelve una copia de la cadena **eliminando los espacios en blanco de ambos extremos de `string`**. No afecta a aquellos que estén en medio.

1	Crear una librería que contenga diversas funciones, las funciones que deberá contener serán de ejercicios de la unidad 2 (algunos será necesario modificarlos un poco), algunas ya tendremos la función hecha, para otras estará el ejercicio dentro del main y será necesario pasarlo a funciones. Para crear la librería creamos una clase estática, cuyos métodos también sean estáticos, de esta forma podremos utilizarlos sin necesidad de crear un objeto, igual que se utiliza la clase Math. Las utilidades mínimas que debe contener son: ◦ Mostrar los números pares del 0 hasta un número que se le pase. (Ejercicio 40). ◦ Mostrar la tabla de multiplicar de un número que se le pase. (Ejercicio 81). ◦ Mostrar los n primeros términos de la serie de Fibonacci, n deberá pasarse como parámetro a la función (Ejercicio 48). ◦ Función que calcule el área de un rectángulo dadas su base y altura que serán números enteros (Ejercicio 79). ◦ Función a la que se le pasan tres parámetros, día, mes y año y nos diga si la fecha es correcta o no. Supondremos que todos los meses tienen 30 días. (Ejercicio 80). Cuando se dice que un dato se le pasa, significa que la clase de utilidades no pedirá ningún dato al usuario, será un método al que se le pase ese valor como argumento, se podrá solicitar el dato al usuario en el main. Crear un programa donde se ejecuten las distintas utilidades de la librería creada. También es posible modificar los ejercicios de la unidad 2, de forma que utilicen la librería.
2	Modificar el ejercicio anterior, para sobrecargar el método de calcular el área de un rectángulo, de forma que se permita que la base y la altura puedan ser de tipo double.
3	Modificar el ejercicio anterior, para sobrecargar el método de calcular el área de un rectángulo, de forma que se permita que la base y la altura puedan tener la misma longitud, de esta forma, solo sería necesario uno de los dos parámetros, base o altura.
4	Modificar el ejercicio de la compañía de móviles para sobrecargar el constructor, de forma que si no se le indique ninguna tarifa de móvil tenga por defecto la tarifa Elefante, pero no debemos eliminar el constructor existente que permite indicar la tarifa deseada. Modificar el código principal para utilizar ambos constructores.